Event Storming & SAGA DESIGN PATTERN

Part 2/4 : Event Sourcing and **Distributed Transactions** for Micro Services

Araf Karsh Hamid, Co-Founder / CTO, MetaMagic Global Inc., NJ, USA

Part 1 : Microservices Architecture

Agenda

1 Micro Services Introduction

1. Micro Services Characteristics
2. Micro Services System Design Model
3. Monolithic Vs. Micro Services Architecture
4. SOA Vs. Micro Services Architecture
5. App Scalability Based on Micro Services
6. Design Patterns

2 Event Storming

1. Event Sourcing Intro
2. Domain and Integration Events
3. Event Sourcing & CQRS Implementations
4. Mind Shift
5. Event Storming
6. Event Storming Restaurant Example
7. Event Storming Process map – Concept
8. ESP Example

3 Y Saga? Scalability Requirement

1. ACID Vs. BASE
2. CAP Theorem
3. Distributed Transactions : 2 Phase Commit
4. Scalability Lessons from eBay

4 SAGA & Other Design Patterns

1. SAGA Design Pattern
2. SAGA Features
3. Local SAGA Features
4. SAGA Execution Container (SEC)
5. Let-it-Crash Design Pattern

5 Case Studies – Examples

1. Handling Invariants
2. Use Case : Travel Booking – SEC
3. Use Case : Travel Booking – Rollback
4. Use Case : Restaurant – Forward Recovery
5. Use Case : Shopping Site – ES / CQRS
6. Use Case : Movie Booking – ES / CQRS

Micro Services Characteristics

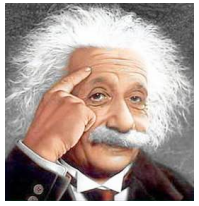
By James Lewis and Martin Fowler

Components via Services

Organized around **Business Capabilities**

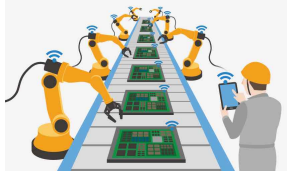
Products NOT Projects

Smart Endpoints & Dumb Pipes



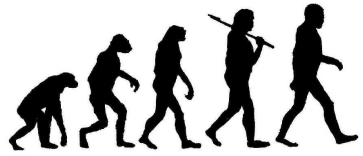
Decentralized Governance & Data Management

Infrastructure **Automation**



Design for **Failure**

Evolutionary Design



The key in making great and growable systems is much more to design how its modules communicate rather than what their internal properties and behaviors should be.
Alan Kay, 1998 email to the Squeak-dev list

Modularity ... is to a technological economy what the division of labor is to a manufacturing one.
W. Brian Arthur,
author of *e Nature of Technology*

We can scale our operation independently, maintain unparalleled system availability, and introduce new services quickly without the need for massive reconfiguration. —
Werner Vogels, CTO, Amazon Web Services

Micro Services System Design Model

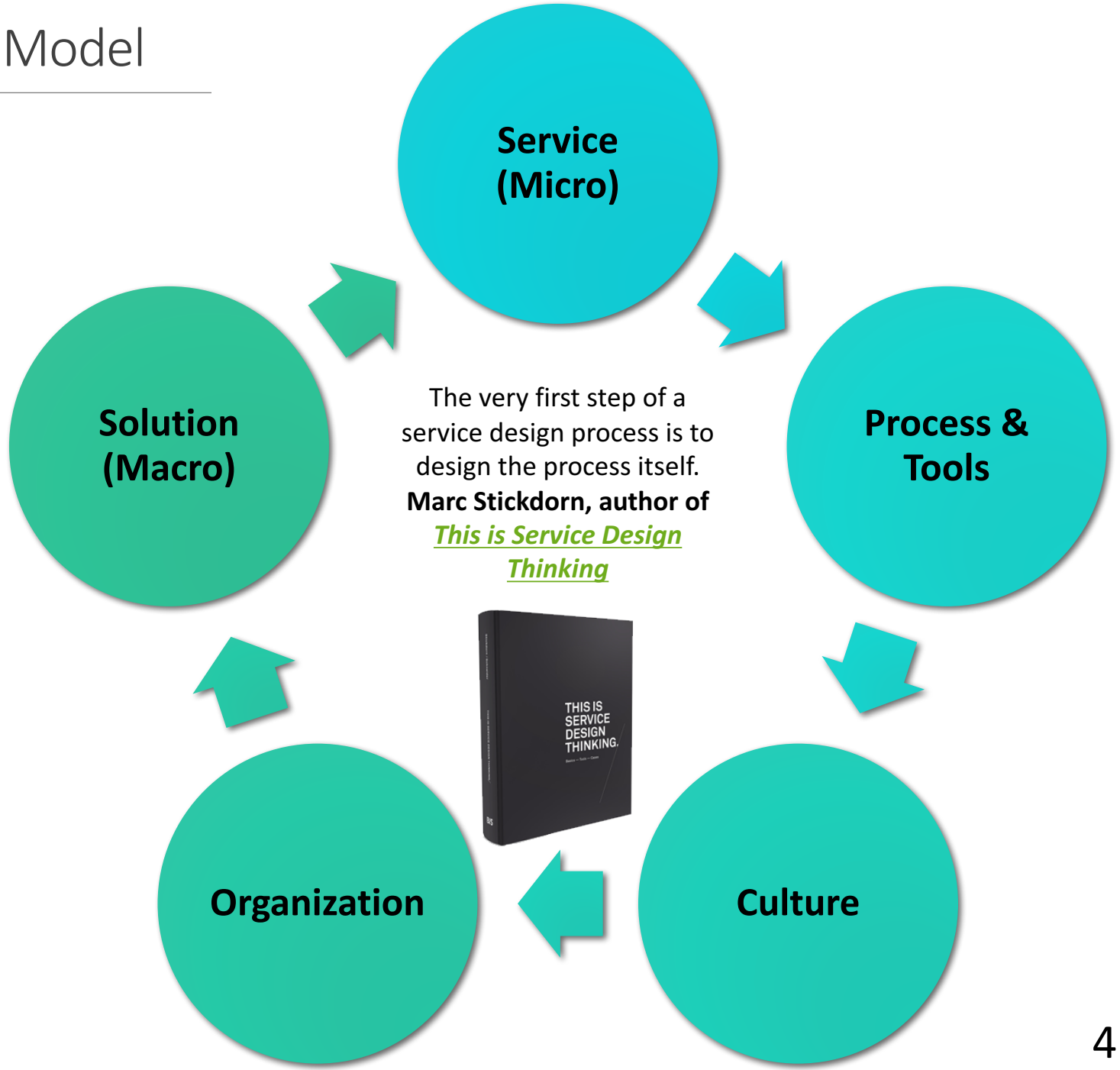
Service: Focuses on a specific Business Capability

Process & Tools: Development, Code Deployment, Maintenance and Product Management

Culture: A Shared set of values, beliefs by everyone. Ubiquitous Language in DDD is an important aspect of Culture.

Organization: Structure, Direction of Authority, Granularity, Composition of Teams.

Solution: Coordinate all inputs and outputs of multiple services. Macro level view of the system allows the designer to focus more on desirable system behavior.



Monolithic vs. Micro Services Example

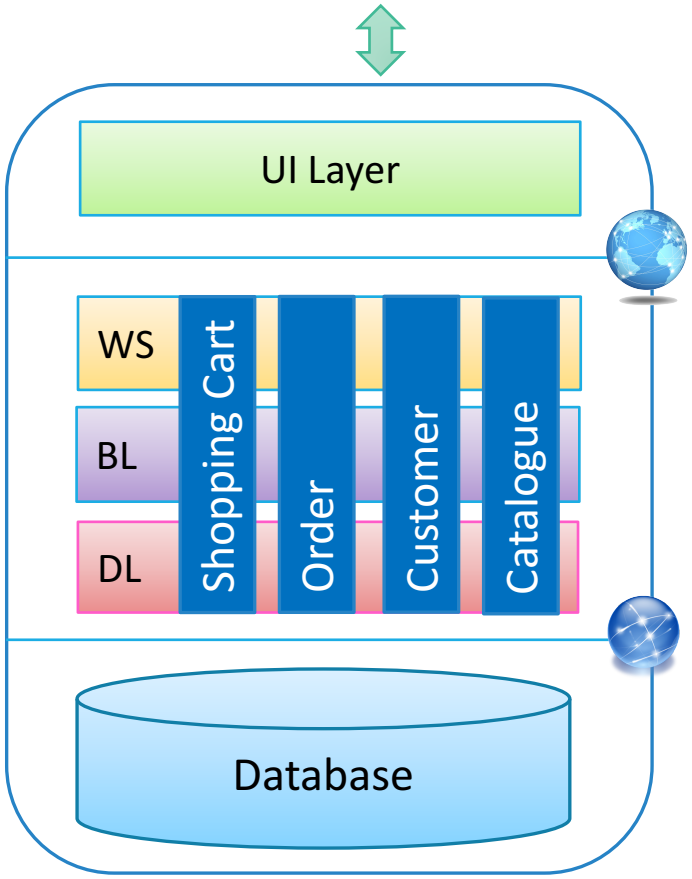




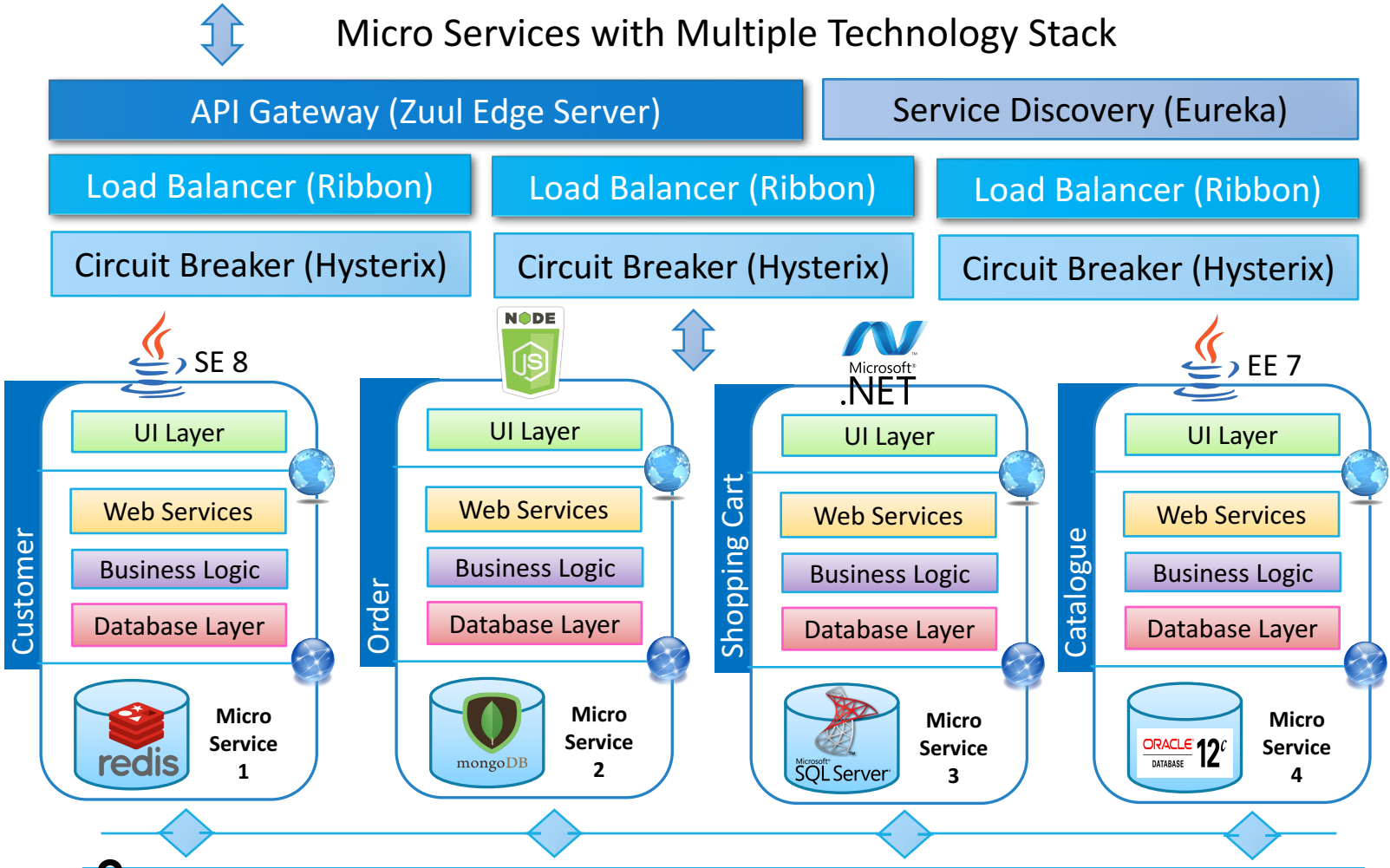
Existing aPaaS vendors creates Monolithic Apps.

This 3 tier model is obsolete now.

Source: Gartner Market Guide for Application Platforms Nov 23, 2016



Traditional Monolithic App using Single Technology Stack



SOA vs. Micro Services Example

Smart Pipes
Lot of Business logic resides in the Pipe

UI Layer

Messaging
REST / SOAP
HTTP
MOM
JMS
ODBC / JDBC
Translation

Enterprise Service Bus

Shopping Cart
Order
Customer
Catalogue

Web Services
XML
WSDL
Addressing
Security
Registry
Management

Producers

Database

Shared Database

Traditional Monolithic App with SOA

Micro Services with Multiple Technology Stack

API Gateway

Service Discovery

Load Balancer | Circuit Breaker

Load Balancer | Circuit Breaker

Load Balancer | Circuit Breaker

SE 8

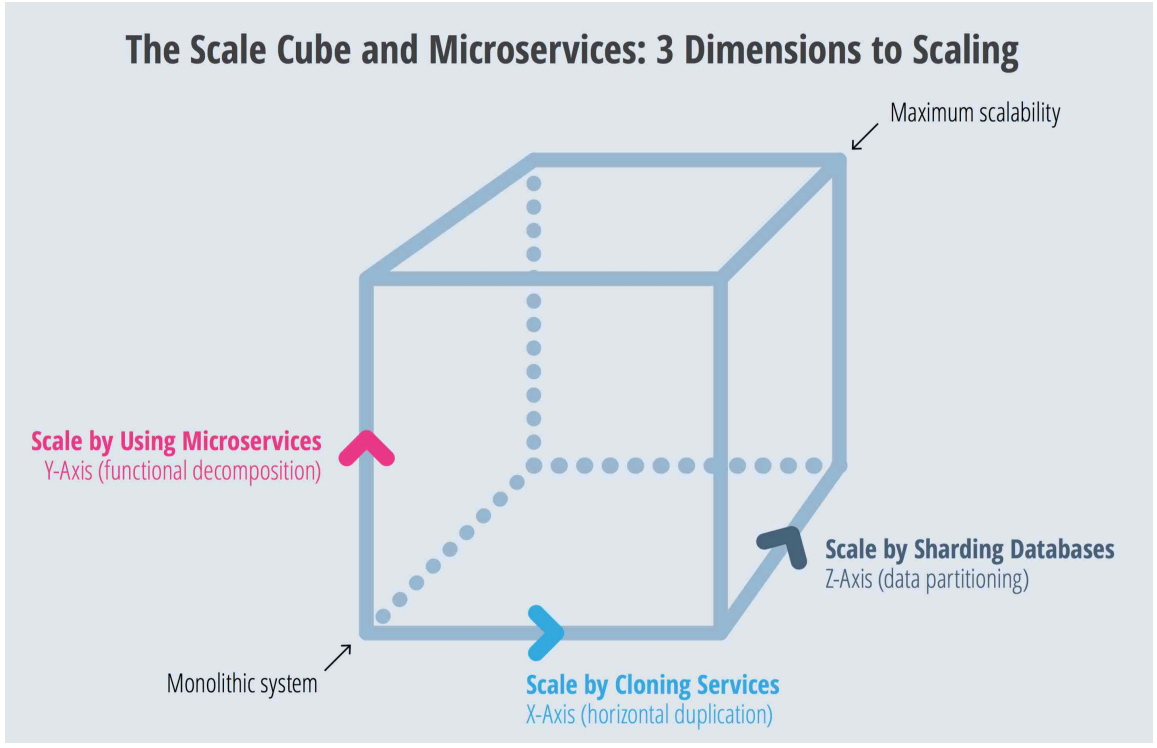
Customer
UI Layer
Web Services
Business Logic
Database Layer
redis
Micro Service 1

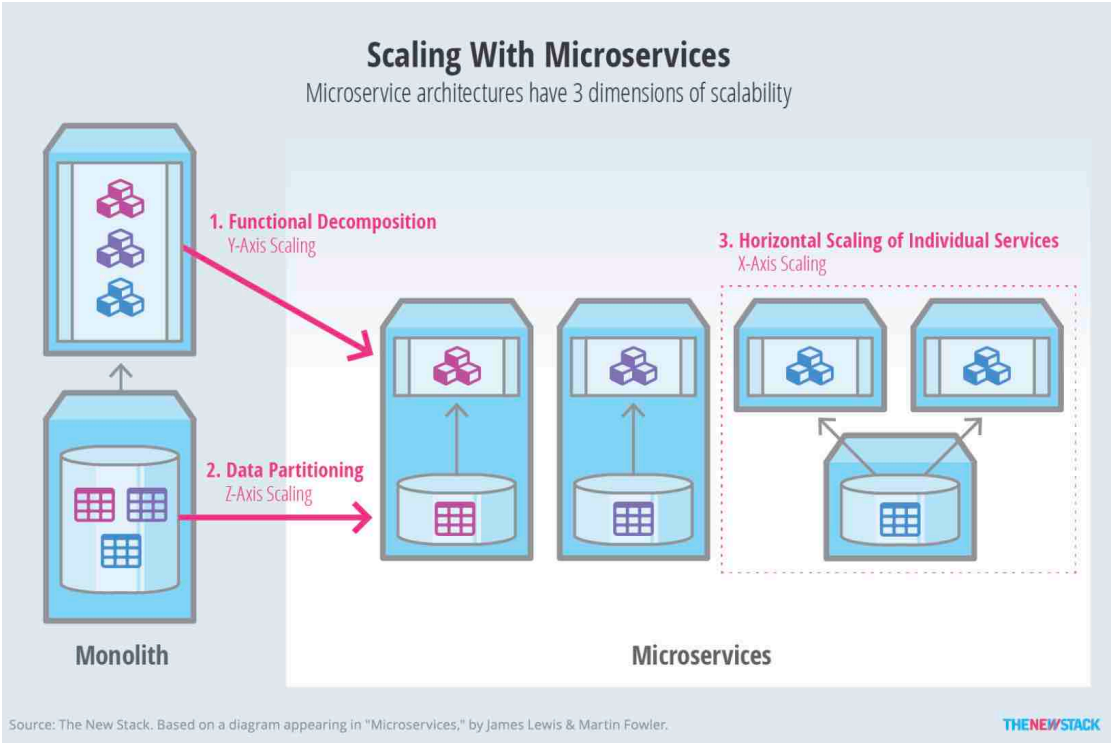
Order
UI Layer
Web Services
Business Logic
Database Layer
mongoDB
Micro Service 2

Shopping Cart
UI Layer
Web Services
Business Logic
Database Layer
Microsoft SQL Server
Micro Service 3

kafka
Event Stream

Scale Cube and Micro Services





1. Y Axis Scaling – Functional Decomposition : Business Function as a Service
2. Z Axis Scaling – Database Partitioning : Avoid locks by Database Sharding
3. X Axis Scaling – Cloning of Individual Services for Specific Service Scalability

Design Patterns

Single Component Pattern

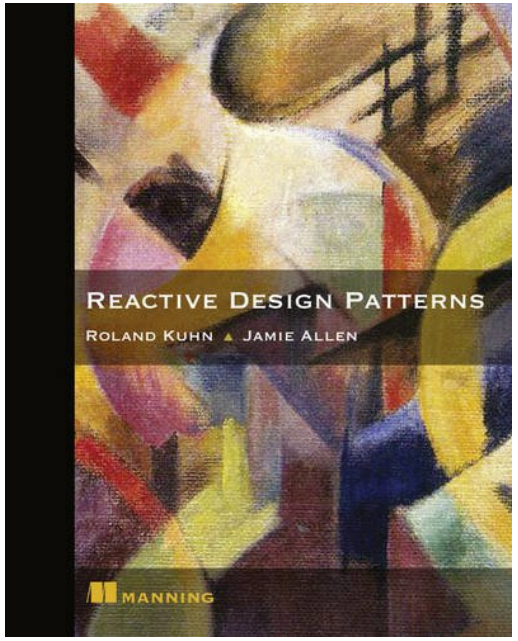
A Component shall do ONLY one thing, But do it in FULL.

Single Responsibility Principle By DeMarco : Structured Analysis & System Specification (Yourdon, New York, 1979)

Saga Pattern

Divide long-lived distributed transactions into quick local ones with compensating actions for recovery.

Pet Helland: Life Beyond Distributed Transactions CIDR 2007

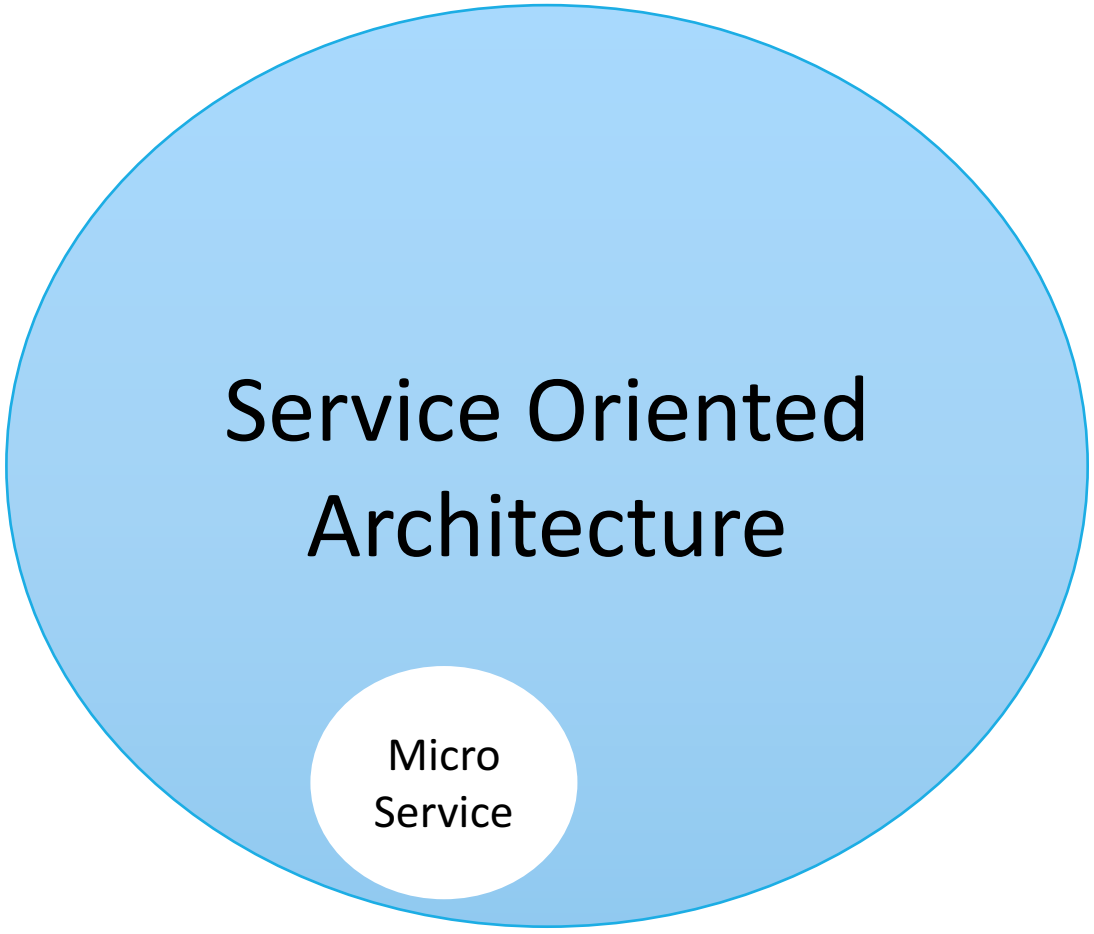


Let-It-Crash Pattern

Prefer a FULL component restart to complex internal failure handling.

Candea & Fox: Crash-Only Software (USENIX HotOS IX, 2003)
Popularized by Netflix Chaos Monkey. Erlang Philosophy

Summary – Micro Services Intro



Benefits

1. Robust
2. Scalable
3. Testable (Local)
4. Easy to Change and Replace
5. Easy to Deploy
6. Technology Agnostic

Martin Fowler – Micro Services Architecture
https://martinfowler.com/articles/microservices.html
Dzone – SOA vs Micro Services : https://dzone.com/articles/microservices-vs-soa-2

Designing Autonomous Teams and Systems

2

Event Storming

1. Event Sourcing Intro
2. Domain Events and Integration Events
3. Event Sourcing & CQRS Implementations
4. Mind Shift
5. Event Storming
6. Event Storming Restaurant Example
7. Event Storming Process map – Concept
8. ESP : Example

Event Sourcing Intro

Standard CRUD Operations – Customer Profile – Aggregate Root

Profile Created | Title Updated | New Address added | Notes Removed

Time T1 | T2 | T3 | T4

Event Sourcing and Derived Aggregate Root

Commands

1. Create Profile
2. Update Title
3. Add Address
4. Delete Notes

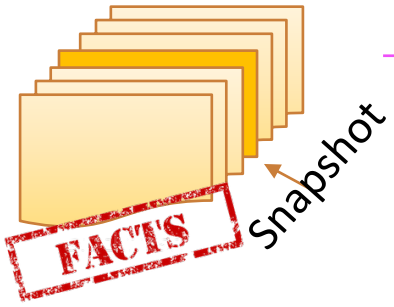
2

Events

1. Profile Created Event
2. Title Updated Event
3. Address Added Event
4. Notes Deleted Event

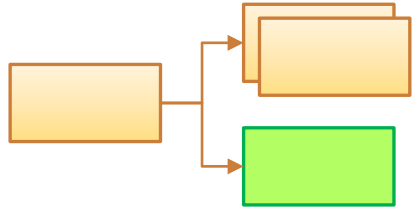
3

Event store



Single Source of Truth

Derived



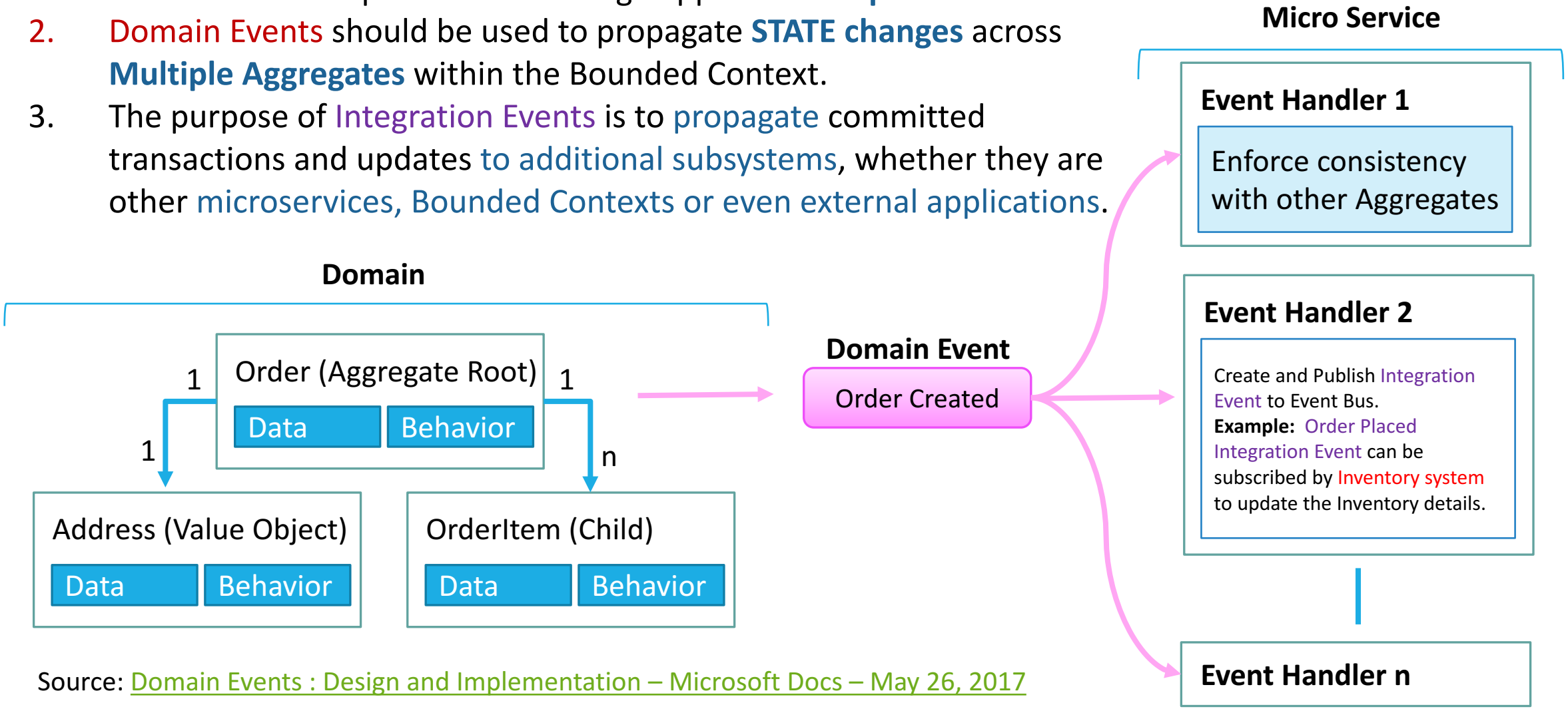
Current State of the Customer Profile

4

Greg Young

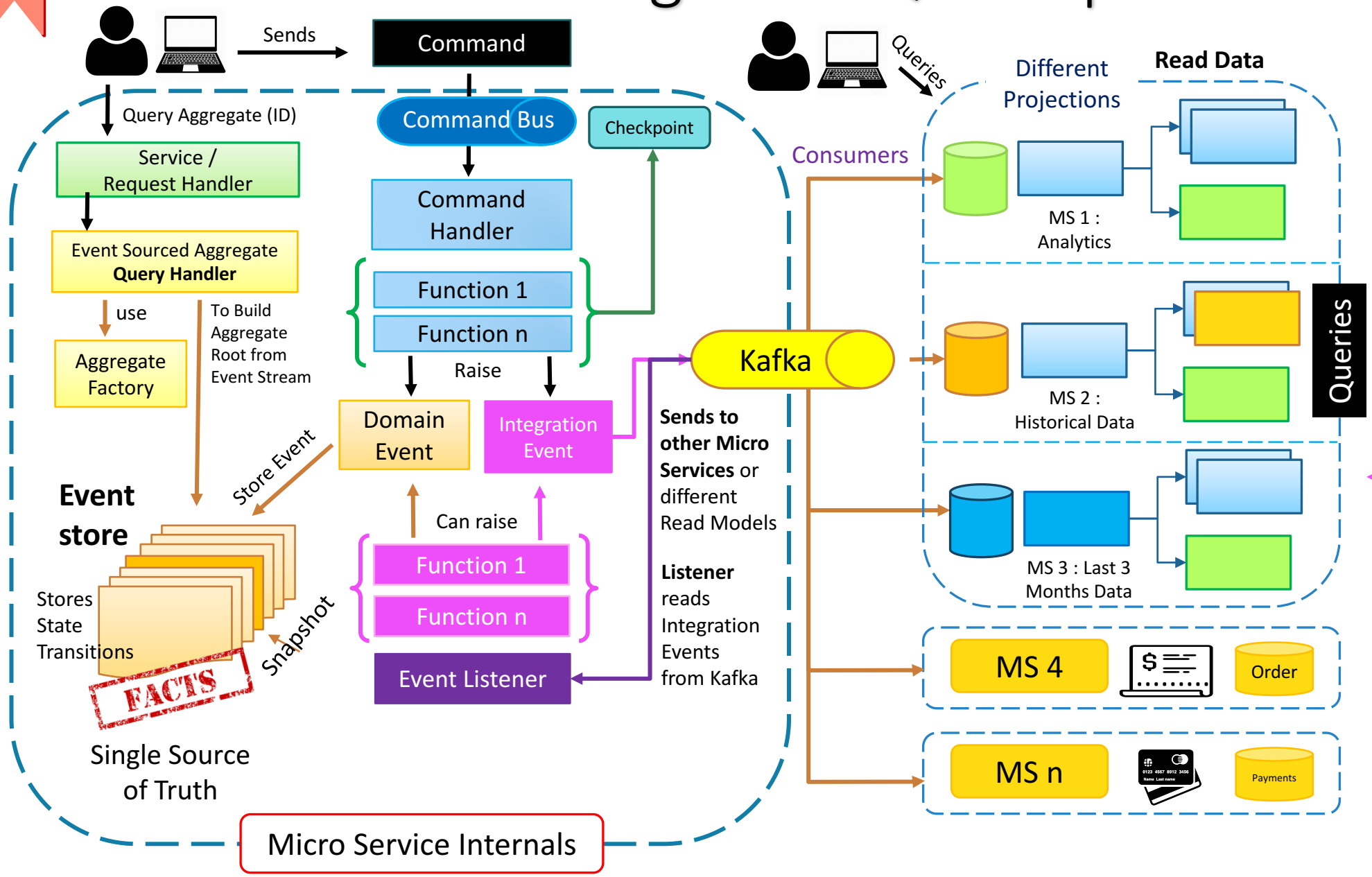
Domain Events & Integration Events

1. Domain Events represent something happened in a **specific Domain**.
2. Domain Events should be used to propagate **STATE changes** across **Multiple Aggregates** within the Bounded Context.
3. The purpose of Integration Events is to propagate committed transactions and updates to additional subsystems, whether they are other microservices, Bounded Contexts or even external applications.



Source: Domain Events : Design and Implementation – Microsoft Docs – May 26, 2017

Event Sourcing and CQRS Implementation



Advantages of ES/CQRS

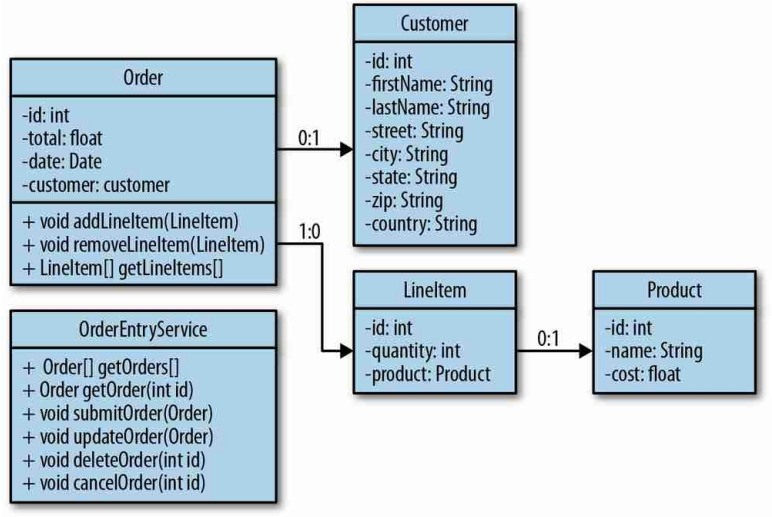
1. Trusted Audit Trail
2. Debugging
3. Historical State
4. Variant Schemas
5. Distribution Support
6. In-Memory Processing
7. Alternative History

2.4

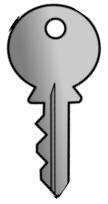
Mind Shift : *From Object Modeling to Process Modeling*

Developers with Strong Object Modeling experience will have trouble making Events a first class citizen.

- How do I start Event Sourcing?
- Where do I Start on Event Sourcing / CQRS?



The Key is:



1. App User's Journey
2. Business Process
3. Ubiquitous Language – DDD
4. Capability Centric Design
5. Outcome Oriented

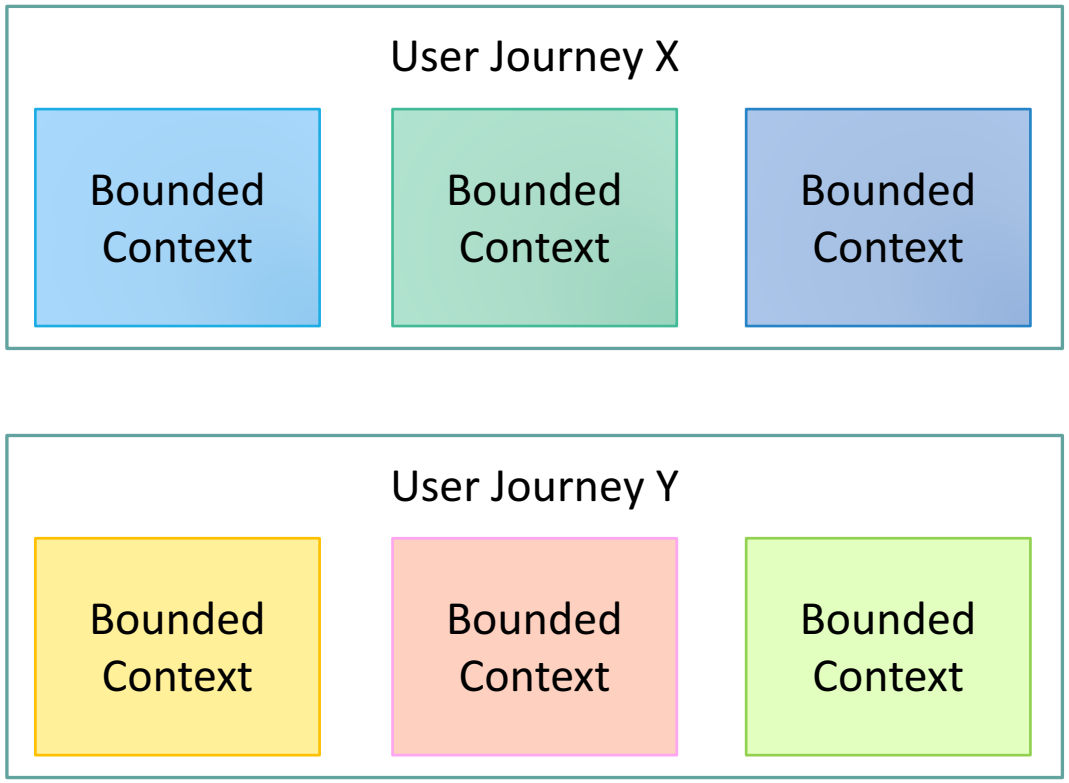
- Think It
- Build It
- Run IT

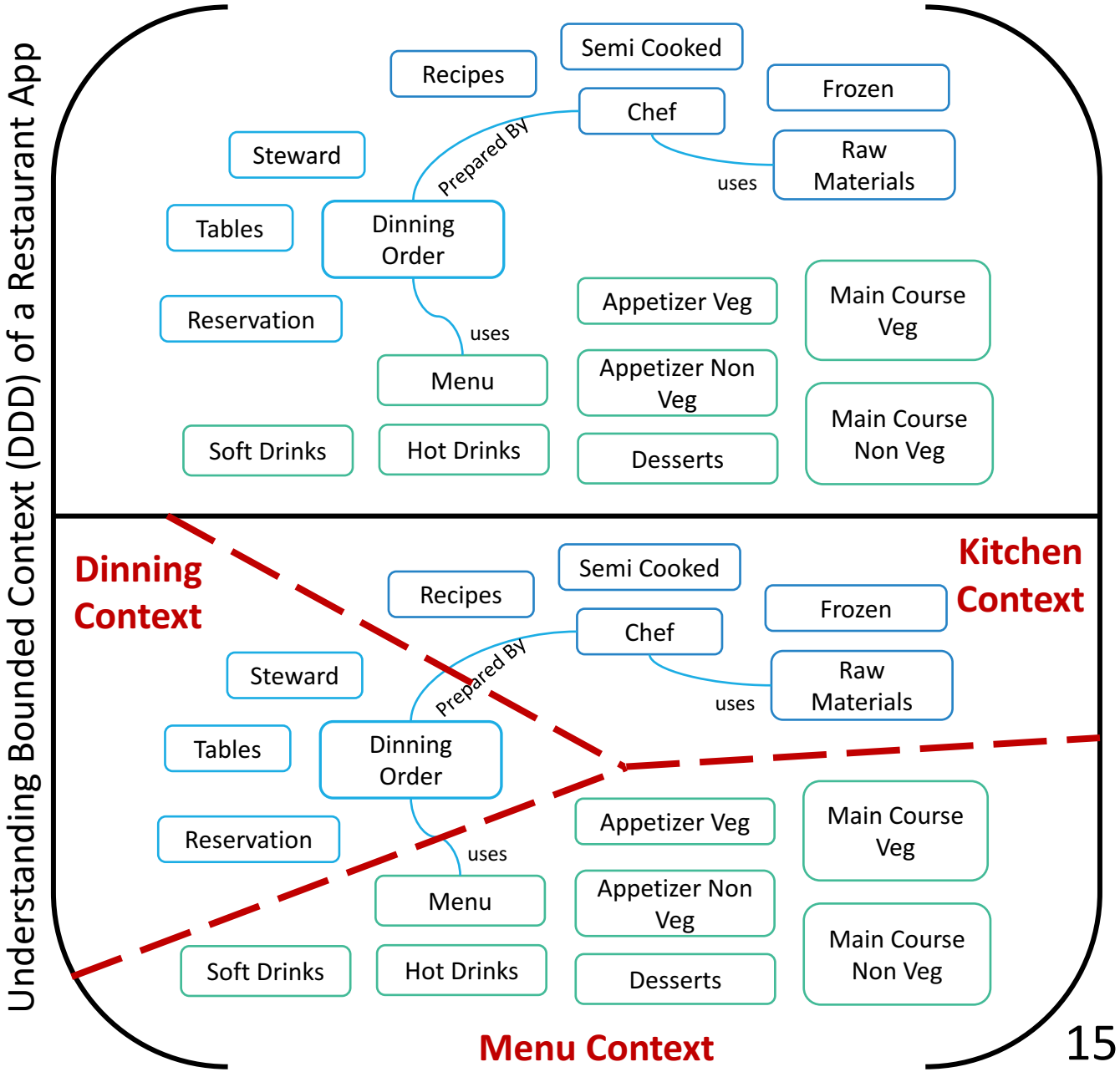
How do you define your End User's Journey & Business Process?

The Best tool to define your process and its tasks.

App User's Journey & Bounded Context (DDD)

An App User's Journey can run across multiple Bounded Context / Micro Services.

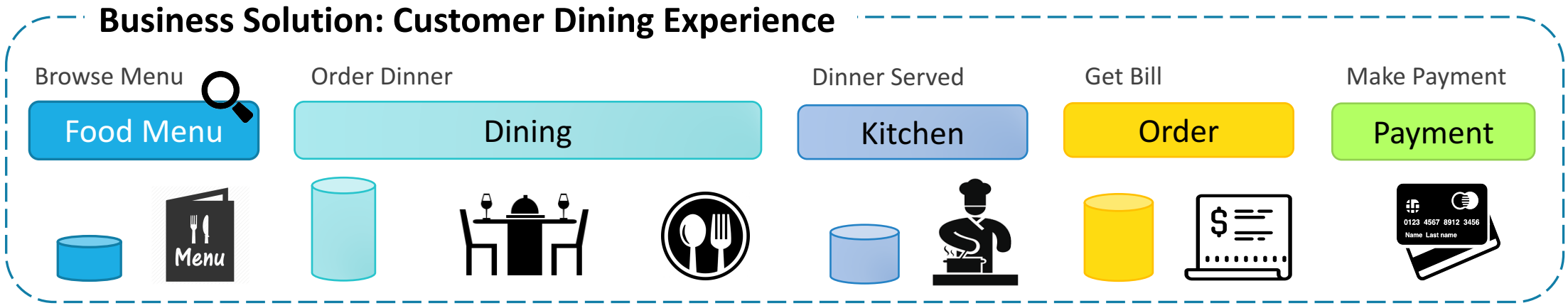




Source: Domain-Driven Design Reference by Eric Evans

Business Solution & Business Process

- Business Solution focuses the entire Journey of the User which can run across multiple Micro Services.
- Business Solution comprises a set of Business Processes.
- Business Process focuses on a specific Micro Service and its functionality.



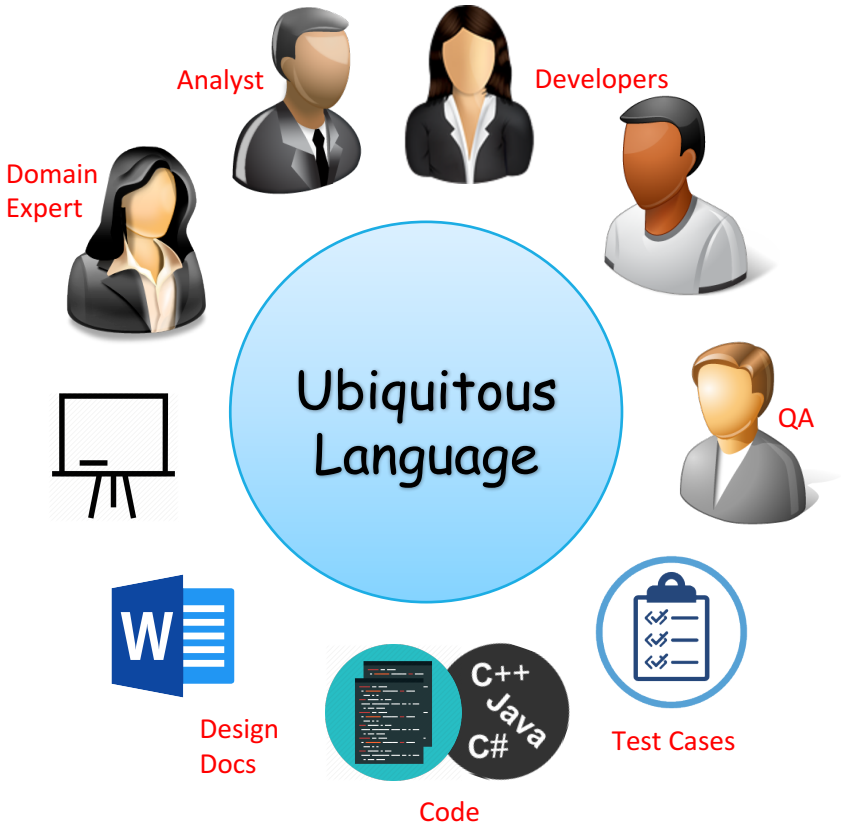
Ubiquitous Language : Understanding Requirement Analysis using DDD

Ubiquitous Language → Vocabulary shared by all involved parties → Used in all forms of spoken / written communication

Food Item :

Eg. Food Item (Navrathnakurma) can have different meaning or properties depends on the context.

- In the Menu Context it's a Veg Dish.
- In the Kitchen Context it's is recipe.
- And in the Dining Context it will have more info related to user feed back etc.



Ubiquitous Language using BDD

As an insurance Broker
I want to know who my Gold Customers are
So that I sell more

Given	Customer John Doe exists
When	he buys insurance ABC for $1000 USD
Then	He becomes a Gold Customer

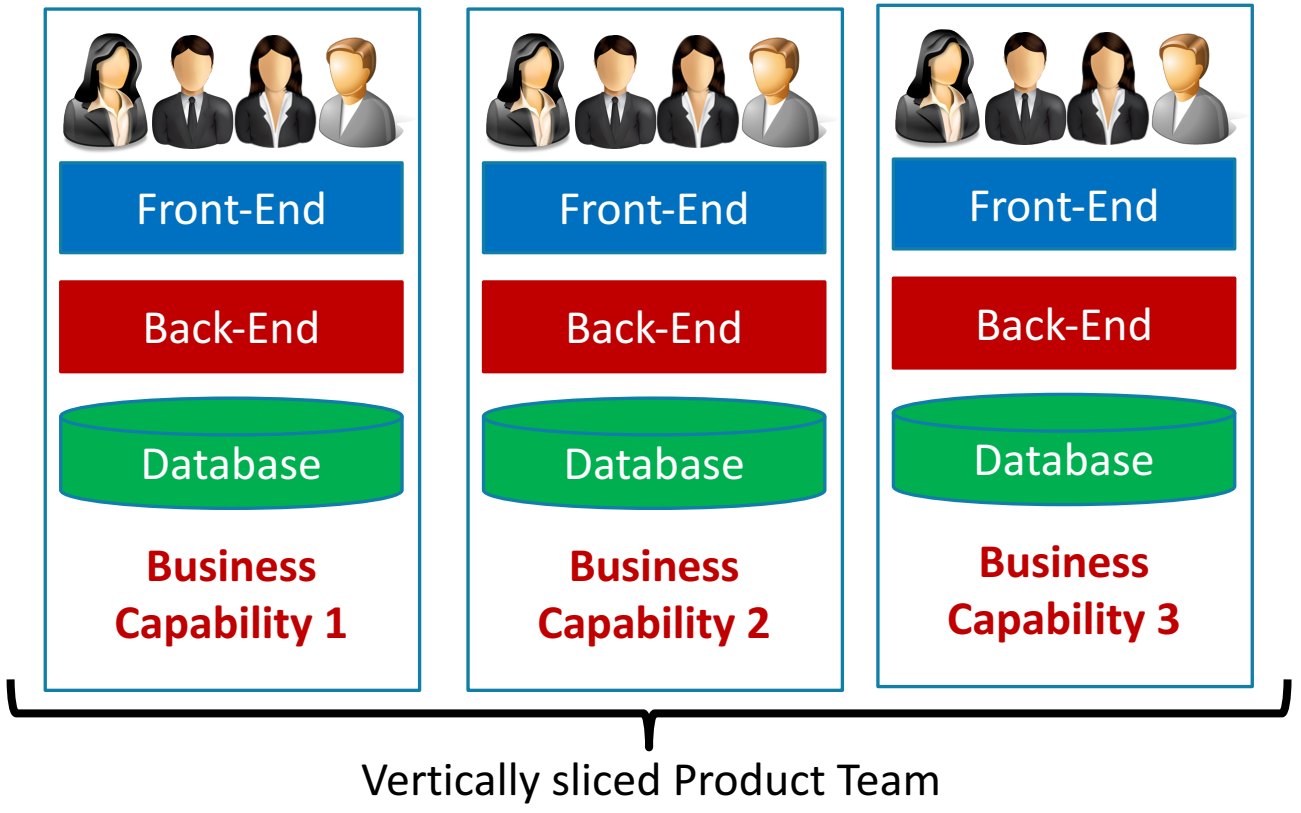
BDD – Behavior Driven Development

Capability Centric Design

In a typical Monolithic way the team is divided based on technology / skill set rather than business functions. This leads to not only bottlenecks but also lack of understanding of the Business Domain.

Business Centric Development

- Focus on Business Capabilities
- Entire team is aligned towards Business Capability.
- From Specs to Operations – The team handles the entire spectrum of Software development.
- Every vertical will have it's own Code Pipeline



Unlocking EVENT STORMING

Event Storming – Concept

Write Data

1. **Process**
   - Define your Business Processes. Eg. Various aspects of Order Processing in an E-Commerce Site, Movie Ticket Booking, Patient visit in Hospital.
2. **Commands**
   - Define the Commands (End-User interaction with your App) to execute the Process. Eg. Add Item to Cart is a Command.
3. **Events**
   - Commands generates the Events to be stored in Event Store. Eg. Item Added Event (in the Shopping Cart).
4. **Event Sourced Aggregate**
   - Current state of the Aggregate is always derived from the Event Store. Eg. Shopping Cart, Order etc. This will be part of the Rich Domain Model (Bounded Context) of the Micro Service.

Read Data

5. **Projections**
   - Projections focuses on the View perspective of the Application. As the Read & Write are different Models, you can have different Projections based on your View perspective.

Event Storming : Restaurant Dining Example – Customer Journey

1 Processes

When people arrive at the Restaurant and take a table, a Table is opened. They may then order drinks and food. Drinks are served immediately by the table staff, however food must be cooked by a chef. Once the chef prepared the food it can then be served. Table is closed when the bill is prepared.

Customer Journey thru Dinning Processes

2 Commands

- Add Drinks
- Add Food
- Update Food
- **Open Table**
- Add Juice
- Add Soda
- Add Appetizer 1
- Add Appetizer 2
- Remove Soda
- Add Food 1
- Add Food 2
- Place Order
- **Close Table**
- Serve Drinks
- Prepare Food
- Serve Food
- **Prepare Bill**
- Process Payment

4 ES Aggregate

- Dinning Order
- Billable Order

Food Menu | Dining | Kitchen | Order | Payment

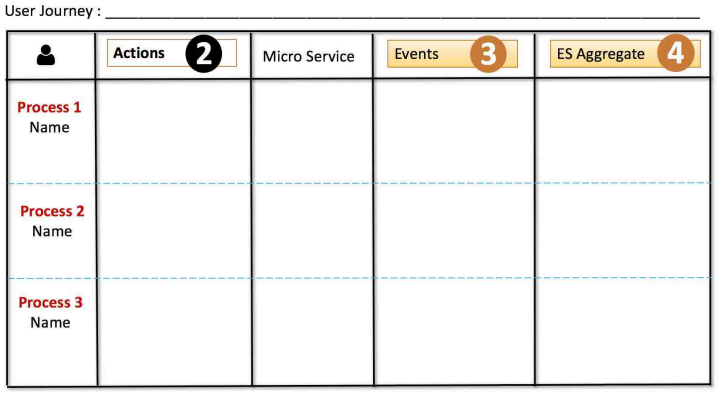
Microservices

3 Events

- Drinks Added
- Food Added
- Food Updated
- Food Discontinued
- Table Opened
- Juice Added
- Soda Added
- Appetizer 1 Added
- Appetizer 2 Added
- Remove Soda
- Food 1 Added
- Food 2 Added
- Order Placed
- Table Closed
- Juice Served
- Soda Served
- Appetizer Served
- Food Prepared
- Food Served
- Bill Prepared
- Payment Processed
- Payment Approved
- Payment Declined
- Cash Paid

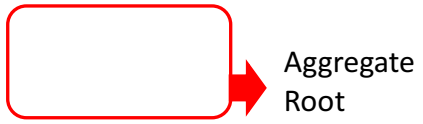
Event Storming Process map – Concept

1. **Define your User Journey.**

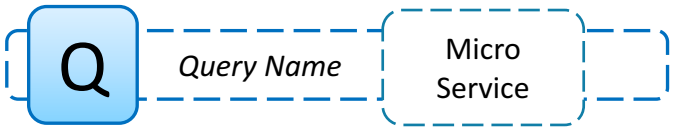


2. **ESP is drawn inside 5 Vertical Swim Lanes.**
   1. Process Name & User interacting with the App
   2. Actions by the user
   3. Micro Service(s) involved
   4. Events raised due to the command
   5. Event Sourced Aggregate Root & Aggregate Factory

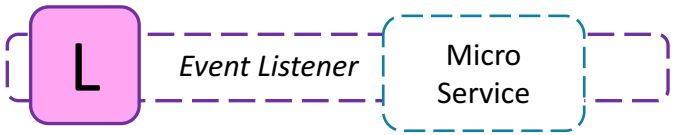
3. **Command & Event(s) Bridge**
   Commands execute the tasks required to complete the Business Process.



4. **Event Sourced Aggregate**
   Event Sourced Aggregate is derived from a Collection of Events & Factory builds these Aggregate Root



5. **Handling Queries in the Process**
   Some commands are issued after the User interaction with Query Result



6. **Listener listens for Domain & Integration Events**
   Listeners plays the key role in Inter Micro Services communications

7. **Events**

Domain Events

Integration Events

8. **Factory**

Factory Builds the Event Sourced Aggregate Root based on the Business Rules defined.

ESP : Customer's Dinning Journey : 1–of–3

	Actions 2	Micro Service	Events 3	ES Aggregate 4
Process 1 Browse Menu 	Q *List Food Menu*	Food Menu 		Aggregate Root is always derived from the Event Store at the time of the Query.
Process 2 Order Food 	C Open Table	Dining 	Table Opened DE	
	C Add Juice		Juice Added DE	
	C Add Soda		Soda Added DE	
	C Add Appetizer 1		Appetizer 1 Added DE	
	C Add Appetizer 2		Appetizer 2 Added DE	
	C Remove Soda		Soda Removed DE	
	C Add Food 1		Food 1 Added DE	
	C Place Order		Order Placed IE	

1. **Dinning Order** (This will have all the Events associated with the Customers Dining experience.)
2. **Order** (Will Contain only the actual order)

Factory Builds the Aggregates

2.8

The purpose of this example is to demonstrate the concept of ES / CQRS thru Event Storming principles.

ESP : Customer's Dinning Journey : 2–of–3

	Actions 2	Micro Service	Events 3	ES Aggregate 4
Process 3 Prepare Food	L Order Listener C Serve Juice C Serve Soda C Serve Appetizer 1 C Prepared Food C Serve Food	Kitchen	Order Placed DE Juice Served DE IE Soda Served DE IE Appetizer 1 Served DE IE Food Prepared DE Food Served DE IE	Aggregate Root is always derived from the Event Store at the time of the Query. 1. **Order Served** (This Aggregate Root will help track the Re-Ordering the food Raw materials)
Process 4 Serve Food	L Serving Listener L Serving Listener L Serving Listener L Serving Listener C Close Table	Dining	Juice Served DE Soda Served DE Appetizer 1 Served DE Food 1 Served DE Table Closed DE	Factory Builds the Aggregates 3. **Billable Order** (This will contain only the Served Item)

ESP : Customer's Dinning Journey : 3–of–3

Process 5
Make Payment

Actions 2

- C — Prepare Bill
- C — Process Payment
- L — Payment Listener
- C — Process Payment

Micro Service

- Order
- Payment

Events 3

- Bill Prepared — DE
- Payment Processed — DE, IE
- Payment Approved — DE
- Payment Status — IE

ES Aggregate 4

1. **Bill** (This will contain only the Served Item)
2. **Bill Payment** (This will contain the Payment details)

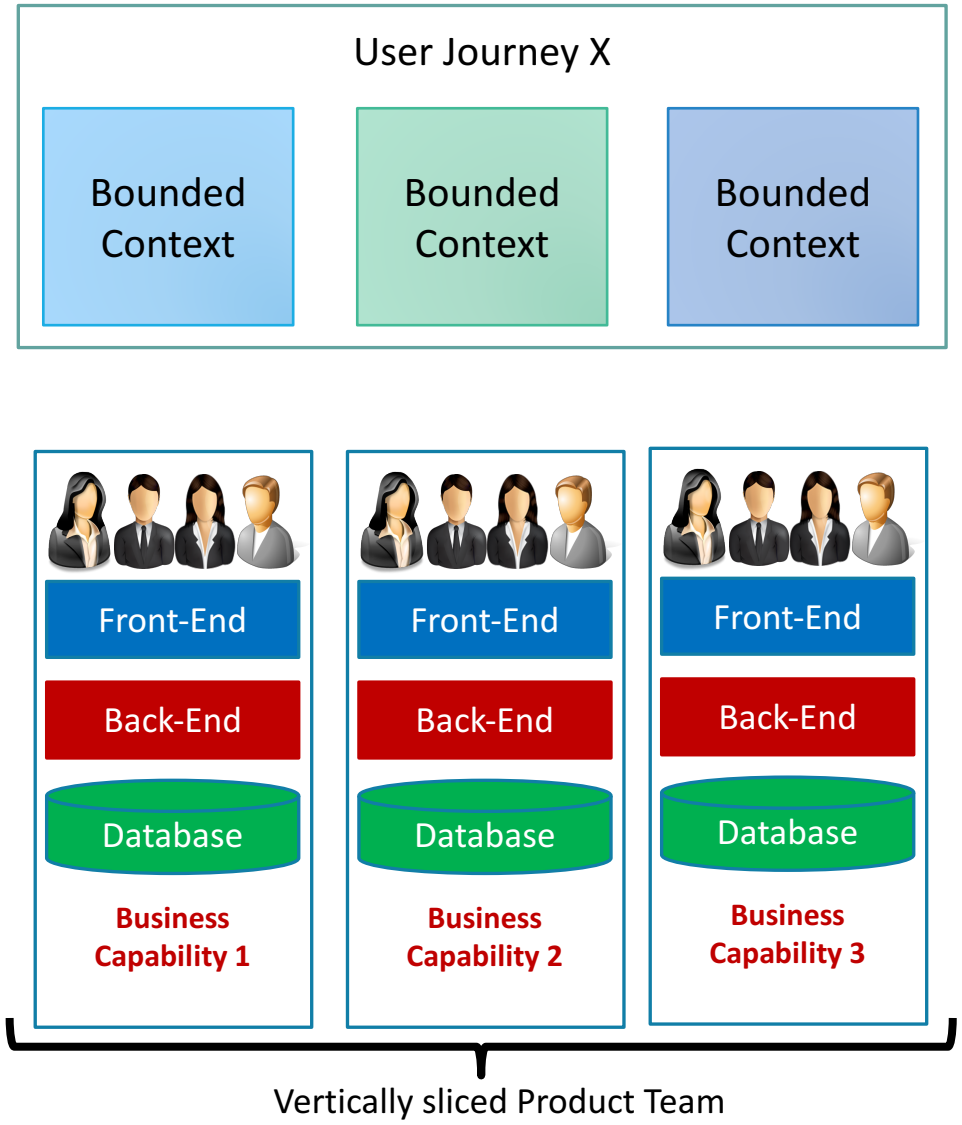
Factory Builds the Aggregates

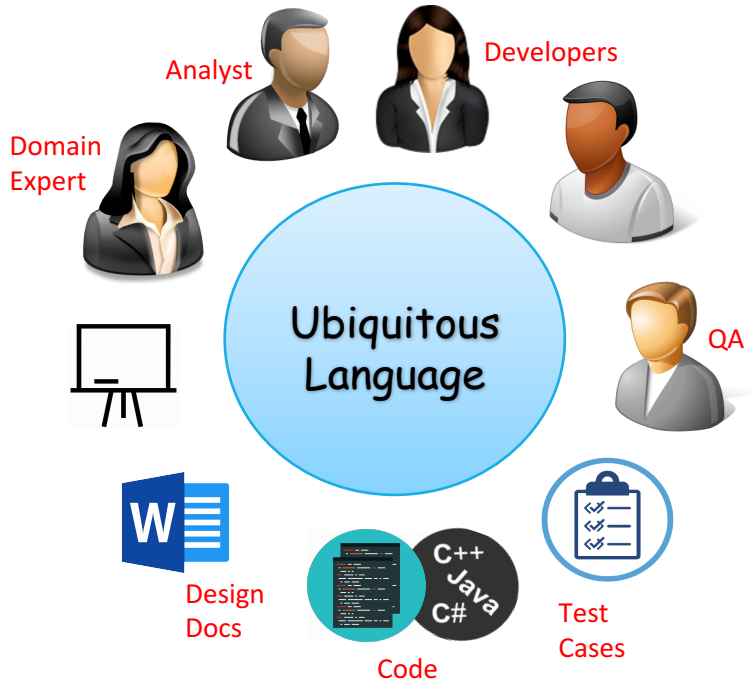
Aggregate Root is always derived from the Event Store at the time of the Query.

2.8

The purpose of this example is to demonstrate the concept of ES / CQRS thru Event Storming principles.

Summary – ESP : Event Sourcing Process map





Y SAGA ?

3

Scalability Requirement in Cloud

1. ACID Vs. BASE
2. CAP Theorem
3. Distributed Transactions : 2 Phase Commit
4. Scalability Lessons from eBay

ACID Vs. BASE

ACID

Atomic

- All operations in a transaction succeed or every operation is rolled back.

Consistent

- On the completion of a transaction, the database is structurally sound.

Isolated

- Transactions do not contend with one another. Contentious access to data is moderated by the database so that transactions appear to run sequentially.

Durable

- The results of applying a transaction are permanent, even in the presence of failures.

BASE

Basic Availability

- The database appears to work most of the time.

Soft-state

- Stores don't have to be write-consistent, nor do different replicas have to be mutually consistent all the time.

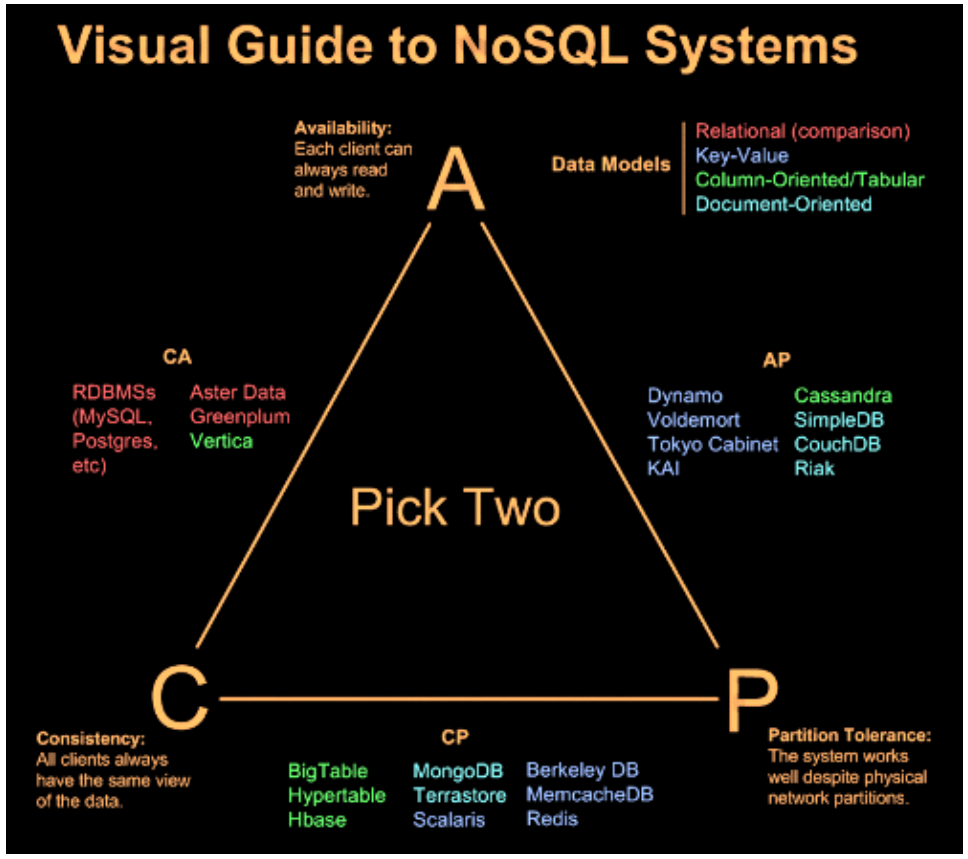
Eventual consistency

- Stores exhibit consistency at some later point (e.g., lazily at read time).

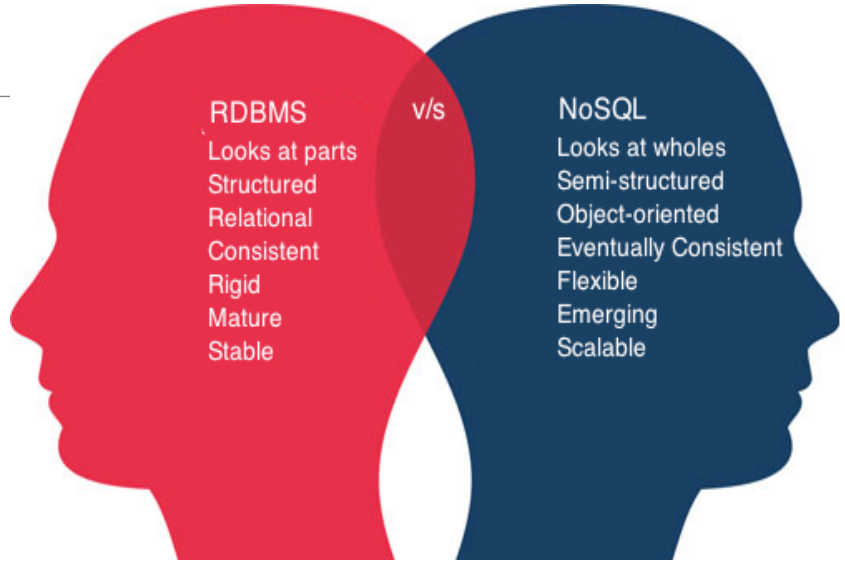
Source: https://neo4j.com/blog/acid-vs-base-consistency-models-explained/

CAP Theorem by Eric Allen Brewer

Pick Any 2!! Say NO to 2 Phase Commit ☺



"In a network subject to communication failures, it is impossible for any web service to implement an atomic read / write shared memory that guarantees a response to every request."



Consistency
Every read receives the most recent write or an error.

Availability
Every request receives a (non-error) response – without guarantee that it contains the most recent write.

Partition Tolerance
The system continues to operate despite an arbitrary number of messages being dropped (or delayed) by the network between nodes.

CAP 12 years later: How the "Rules have changed"

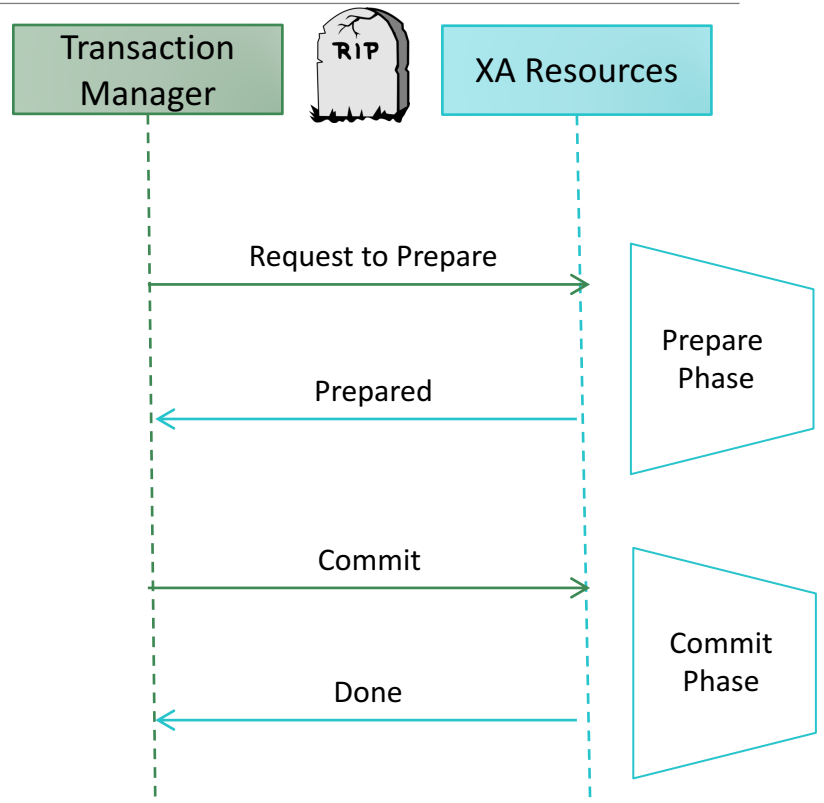
Source: https://en.wikipedia.org/wiki/CAP_theorem | http://en.wikipedia.org/wiki/Eric_Brewer_(scientist)

Distributed Transactions : 2 Phase Commit

2 PC or not 2 PC, Wherefore Art Thou XA?

How does 2PC impact scalability?

- Transactions are committed in two phases.
- This involves communicating with every database (XA Resources) involved to determine if the transaction will commit in the first phase.
- During the second phase each database is asked to complete the commit.
- While all of this coordination is going on, locks in all of the data sources are being held.
- ***The longer duration locks create the risk of higher contention.***
- *Additionally,* ***the two phases require more database processing time than a single phase commit.***
- **The result is lower overall TPS in the system.**



Solution : Resilient System

- Event Based
- Design for failure
- Asynchronous Recovery
- Make all operations idempotent.
- Each DB operation is a 1 PC

Source : Pat Helland (Amazon) : Life Beyond Distributed Transactions Distributed Computing : http://dancres.github.io/Pages/

Scalability Best Practices : Lessons from ebay

	Best Practices	Highlights
#1	Partition By Function	• Decouple the Unrelated Functionalities. • Selling functionality is served by one set of applications, bidding by another, search by yet another. • 16,000 App Servers in 220 different pools • 1000 logical databases, 400 physical hosts
#2	Split Horizontally	• Break the workload into manageable units. • eBay's interactions are stateless by design • All App Servers are treated equal and none retains any transactional state • Data Partitioning based on specific requirements
#3	Avoid Distributed Transactions	• 2 Phase Commit is a pessimistic approach comes with a big COST • **CAP Theorem** (Consistency, Availability, Partition Tolerance). Apply any two at any point in time. • @ eBay No Distributed Transactions of any kind and NO 2 Phase Commit.
#4	Decouple Functions Asynchronously	• If Component A calls component B synchronously, then they are tightly coupled. For such systems to scale A you need to scale B also. • If Asynchronous A can move forward irrespective of the state of B • SEDA (Staged Event Driven Architecture)
#5	Move Processing to Asynchronous Flow	• Move as much processing towards Asynchronous side • Anything that can wait should wait
#6	Virtualize at All Levels	• Virtualize everything. eBay created their on O/R layer for abstraction
#7	Cache Appropriately	• Cache Slow changing, read-mostly data, meta data, configuration and static data.

Source: http://www.infoq.com/articles/ebay-scalability-best-practices

Summary : Scalability Requirement in Cloud

1. Availability and Partition Tolerance is more important than immediate Consistency.
2. Eventual Consistency is more suitable in a highly scalable Cloud Environment
3. Two Phase Commit has its limitations from Scalability perspective and it's a Single Point of Failure.
4. Scalability examples from eBay, Amazon, Netflix, Uber, Airbnb etc.

4

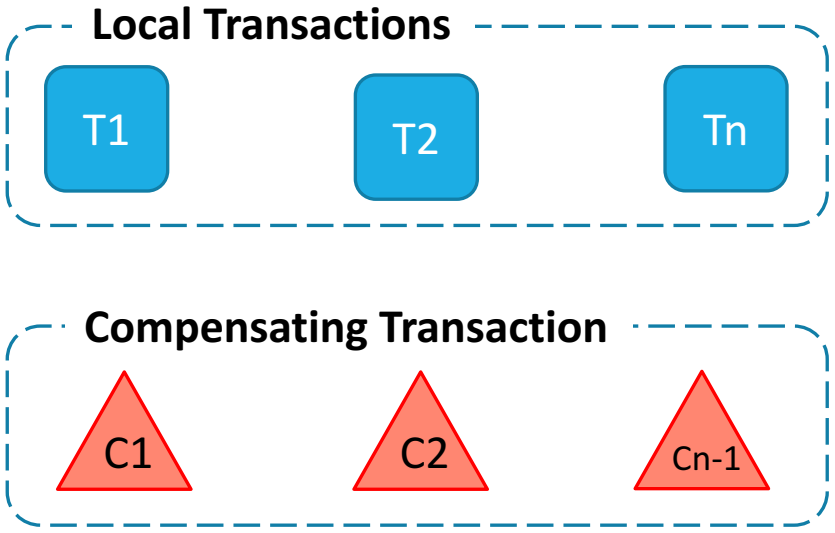
SAGA & Other Design Patterns

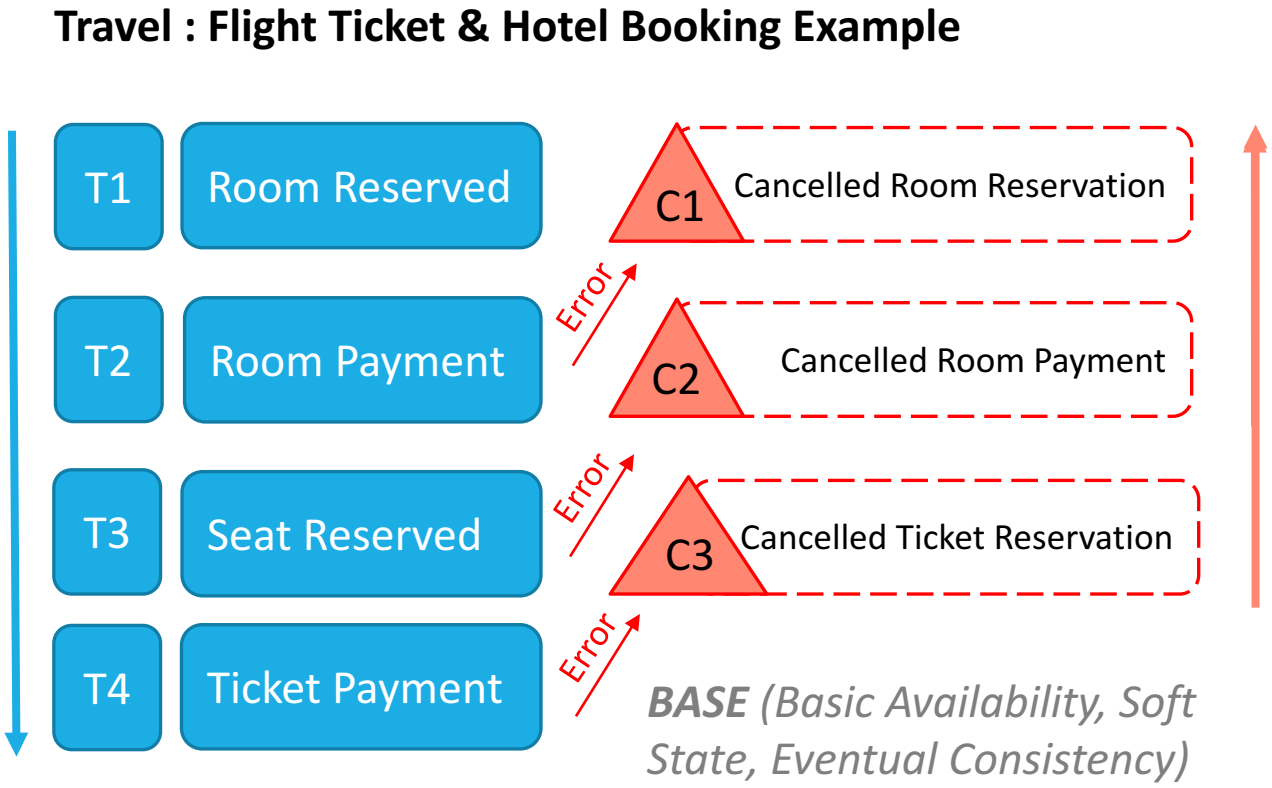
1. SAGA Introduction
2. SAGA Features
3. Local SAGA Features
4. SAGA Execution Container (SEC)
5. Let-it-Crash Design Pattern
6. Domain Events & Integration Events

SAGA Design Pattern Introduction

Long Lived Transactions (LLTs) *hold on to DB resources for relatively long periods of time, significantly delaying the termination of shorter and more common transactions.*

Divide long–lived, distributed transactions into quick local ones with compensating actions for recovery.





Source: SAGAS (1987) Hector Garcia Molina / Kenneth Salem, Dept. of Computer Science, Princeton University, NJ, USA

SAGA Features

Type	Examples
1. Backward Recovery (Rollback) T_1 T_2 T_3 T_4 C_3 C_2 C_1	Order Processing, Banking Transactions, Ticket Booking
2. Forward Recovery with Save Points T_1 (sp) T_2 (sp) T_3 (sp)	Updating individual scores in a Team Game.

- To recover from Hardware Failures, SAGA needs to be persistent.
- Save Points are available for both Forward and Backward Recovery.

Source: SAGAS (1987) Hector Garcia Molina / Kenneth Salem, Dept. of Computer Science, Princeton University, NJ, USA

Local SAGA Features

1. Part of the Micro Services
2. Local Transactions and Compensation Transactions
3. SAGA State is persisted
4. All the Local transactions are based on Single Phase Commit (1 PC)
5. Developers need to ensure that appropriate compensating transactions are Raised in the event of a failure.

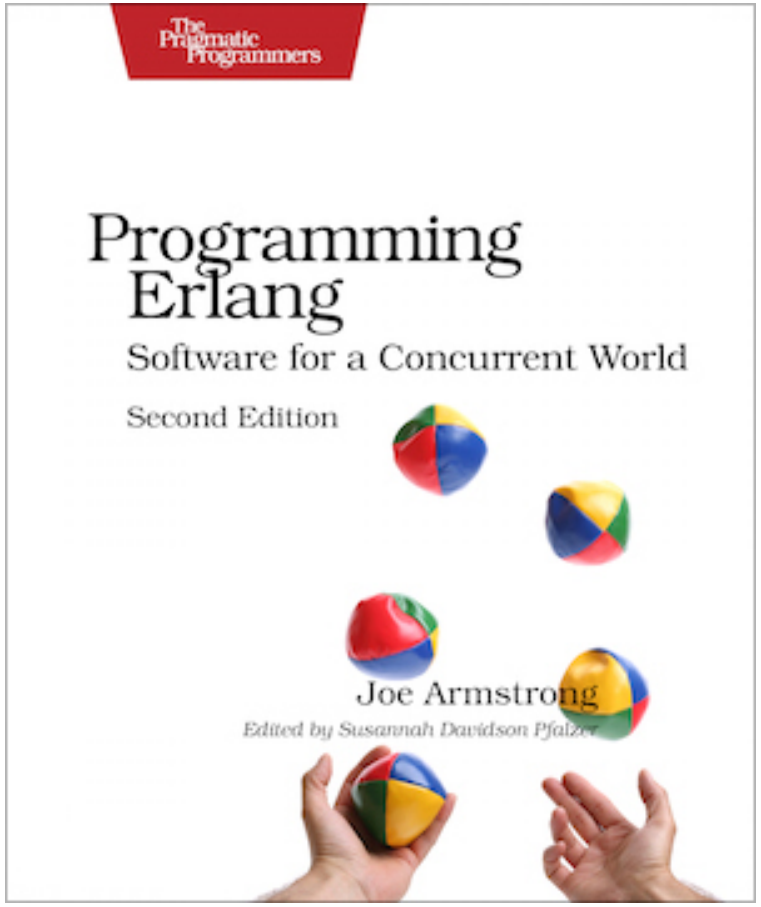
API Examples

```
@StartSaga(name="HotelBooking")
public void reserveRoom(…) {
}

@EndSaga(name="HotelBooking")
public void payForTickets(…) {
}

@AbortSaga(name="HotelBooking")
public void cancelBooking(…) {
}

@CompensationTx()
public void cancelReservation(…) {
}
```

SAGA Execution Container

1. SEC is a separate Process
2. Stateless in nature and Saga state is stored in a messaging system (Kafka is a Good choice).
3. SEC process failure MUST not affect Saga Execution as the restart of the SEC must start from where the Saga left.
4. SEC – No Single Point of Failure (Master Slave Model).
5. Distributed SAGA Rules are defined using a DSL.

Let-it-Crash Design Pattern – Erlang Philosophy



- The Erlang view of the world is that everything is a process and that processes can interact only by exchanging messages.
- A typical Erlang program might have hundreds, thousands, or even millions of processes.
- Letting processes crash is central to Erlang. It's the equivalent of unplugging your router and plugging it back in – as long as you can get back to a known state, this turns out to be a very good strategy.
- To make that happen, you build supervision trees.
- A supervisor will decide how to deal with a crashed process. It will restart the process, or possibly kill some other processes, or crash and let someone else deal with it.
- Two models of concurrency: Shared State Concurrency, & Message Passing Concurrency. The programming world went one way (toward shared state). The Erlang community went the other way.
- All languages such as C, Java, C++, and so on, have the notion that there is this stuff called state and that we can change it. The moment you share something you need to bring Mutex a Locking Mechanism.
- Erlang has no mutable data structures (that's not quite true, but it's true enough). No mutable data structures = No locks. No mutable data structures = Easy to parallelize.

Let-it-Crash Design Pattern

1. The idea of Messages as the first class citizens of a system, has been rediscovered by the Event Sourcing / CQRS community, along with a strong focus on domain models.
2. Event Sourced Aggregates are a way to Model the Processes and NOT things.
3. Each component MUST tolerate a crash and restart at any point in time.
4. All interaction between the components must tolerate that peers can crash. This mean ubiquitous use of timeouts and Circuit Breaker.
5. Each component must be strongly encapsulated so that failures are fully contained and cannot spread.
6. All requests sent to a component MUST be self describing as is practical so that processing can resume with a little recovery cost as possible after a restart.

4.5

Let-it-Crash : Comparison Erlang Vs. Micro Services Vs. Monolithic Apps

		Erlang Philosophy	Micro Services Architecture	Monolithic Apps (Java, C++, C#, Node JS ...)
1	Perspective	Everything is a Process	Event Sourced Aggregates are a way to model the Process and NOT things.	Things (defined as Objects) and Behaviors
2	Crash Recovery	Supervisor will decide how to handle the crashed process	A combination of IaaS (AWS / Azure …), Load Balancer (Netflix Ribbon) and Service Discovery (Netflix Eureka) will handle the crashed Micro Service along with Circuit Breaker pattern to handle fallback mechanism.	Not available. Most of the monolithic Apps are Stateful and Crash Recovery needs to be handled manually and all languages other than Erlang focuses on defensive programming.
3	Concurrency	Message Passing Concurrency	Domain Events for state changes within a Bounded Context & Integration Events for external Systems.	Mostly Shared State Concurrency
4	State	Stateless : Mostly Immutable Structures	Immutability is handled thru Event Sourcing along with Domain Events and Integration Events.	Predominantly Stateful with Mutable structures and Mutex as a Locking Mechanism
5	Citizen	Messages	Messages are 1st class citizen by Event Sourcing / CQRS pattern with a strong focus on Domain Models	Mutable Objects and Strong focus on Domain Models and synchronous communication.

Summary : Saga & Other Design Patterns

1. Saga solves the distributed Transaction problems 2 Phase commit had and with its asynchronous and not having a single point of failure makes it a perfect candidate for cloud computing.

2. Developer Complexity is increased with Saga Pattern as Developers needs to focus on Rollbacks and Roll forwards with Compensating transactions.

3. Let-it-Crash is not just a programming pattern like Singleton, Factory or Observer etc. It’s a pattern that runs across the programming paradigm and your infrastructure paradigm.

5

Case Studies – Examples

1. Handling Invariants
2. Use Case : Travel Booking – SEC
3. Use Case : Travel Booking – Rollback
4. Use Case : Restaurant – Forward Recovery
5. Use Case : Shopping Site – Event Sourcing / CQRS
6. Use Case : Movie Booking – Event Sourcing / CQRS
7. Use Case : Restaurant Dinning – Event Sourcing / CQRS
8. Summary
9. References

Handling Invariants – Monolithic to Micro Services

In a typical Monolithic App Customer Credit Limit info and the order processing is part of the same App. Following is a typical pseudo code.

In Micro Services world with Event Sourcing, it's a distributed environment. The order is cancelled if the Credit is NOT available. If the Payment Processing is failed then the Credit Reserved is cancelled.

Monolithic 2 Phase Commit

Begin Transaction

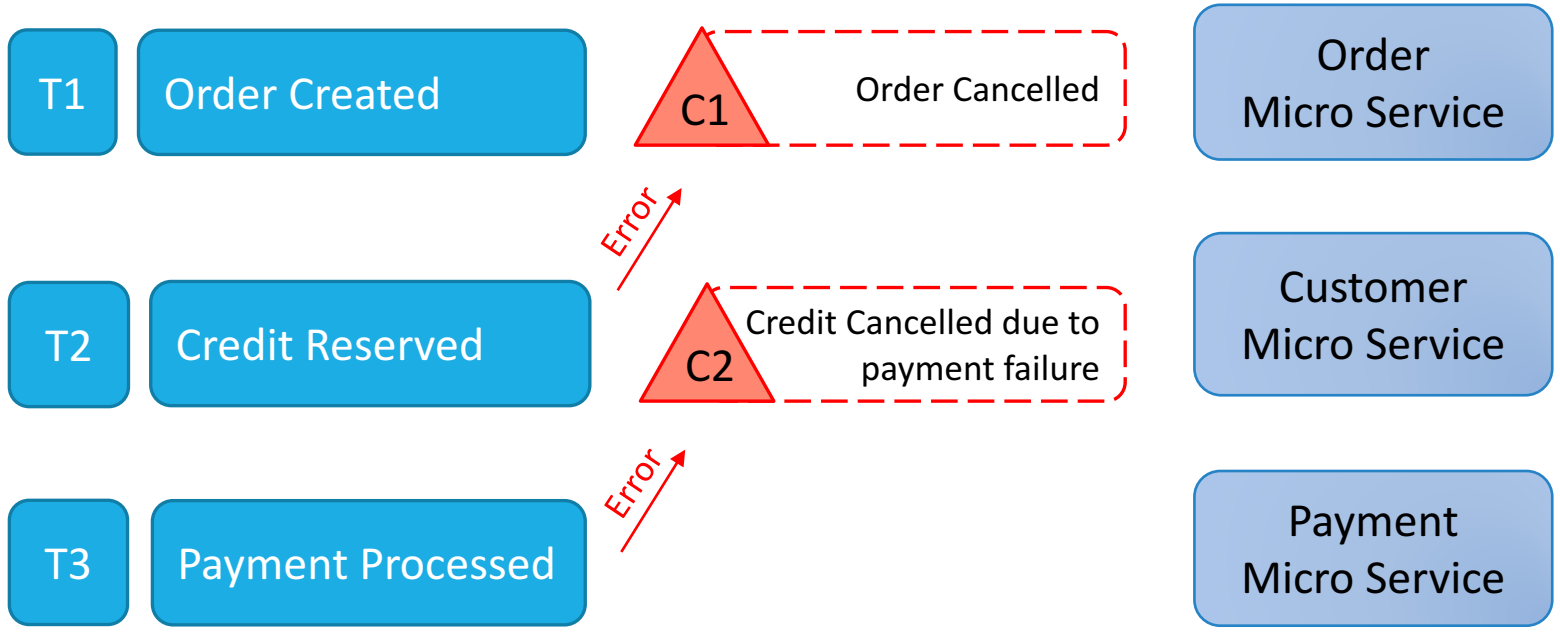
If Order Value <= Available Credit

Process Order

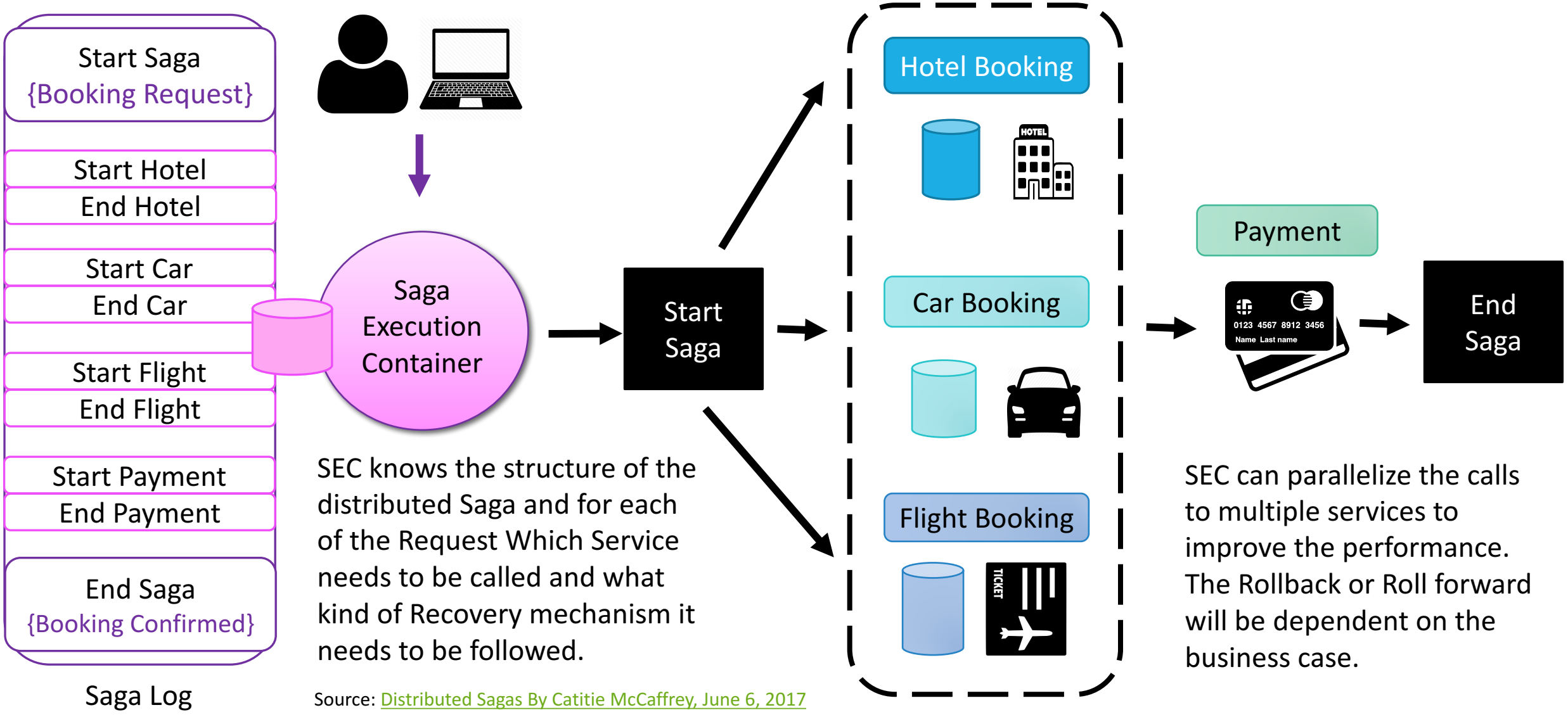
Process Payments

End Transaction

https://en.wikipedia.org/wiki/Invariant_(computer_science)



Use Case : Travel Booking – Distributed Saga (SEC)

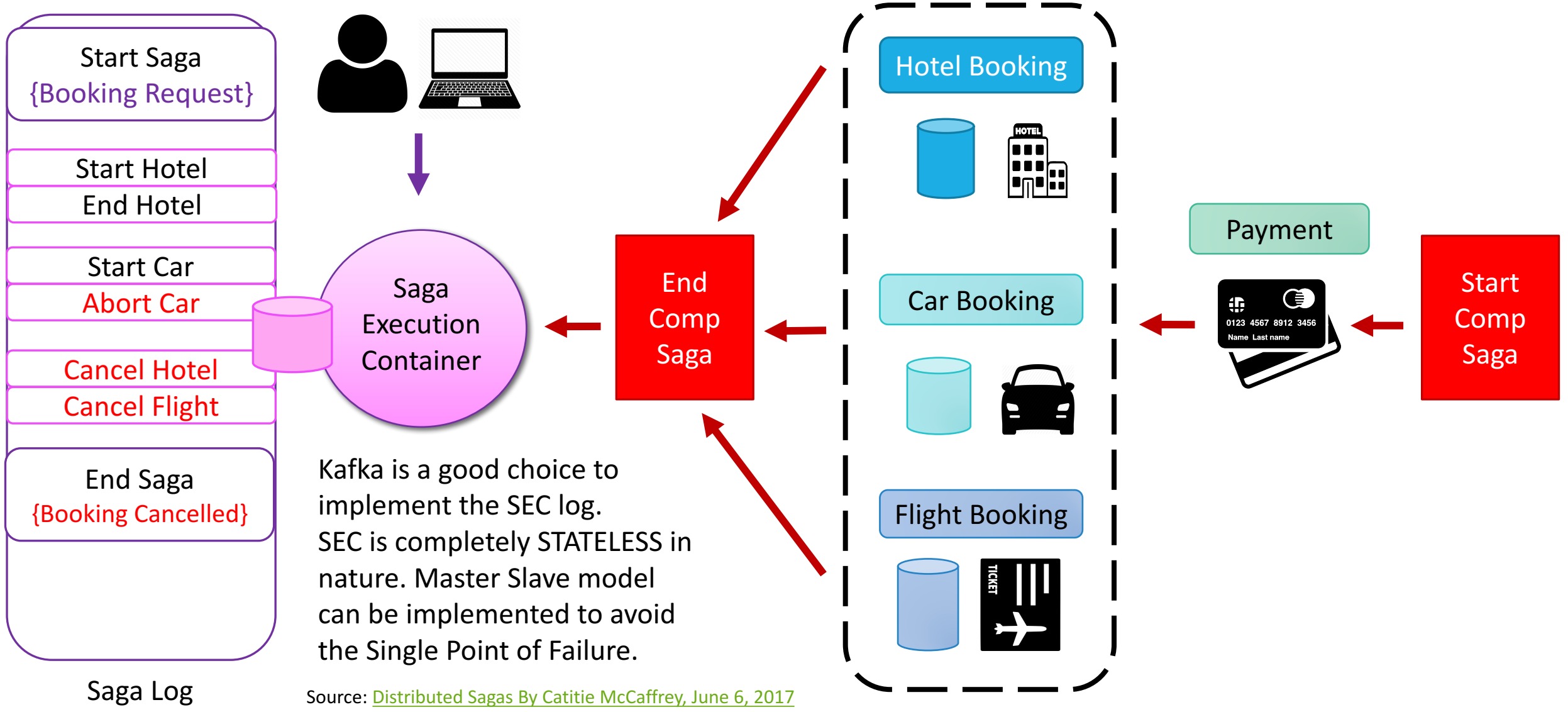


SEC knows the structure of the distributed Saga and for each of the Request Which Service needs to be called and what kind of Recovery mechanism it needs to be followed.

SEC can parallelize the calls to multiple services to improve the performance. The Rollback or Roll forward will be dependent on the business case.

Source: Distributed Sagas By Catitie McCaffrey, June 6, 2017

Use Case : Travel Booking – Rollback



Kafka is a good choice to implement the SEC log.
SEC is completely STATELESS in nature. Master Slave model can be implemented to avoid the Single Point of Failure.

Source: Distributed Sagas By Catitie McCaffrey, June 6, 2017

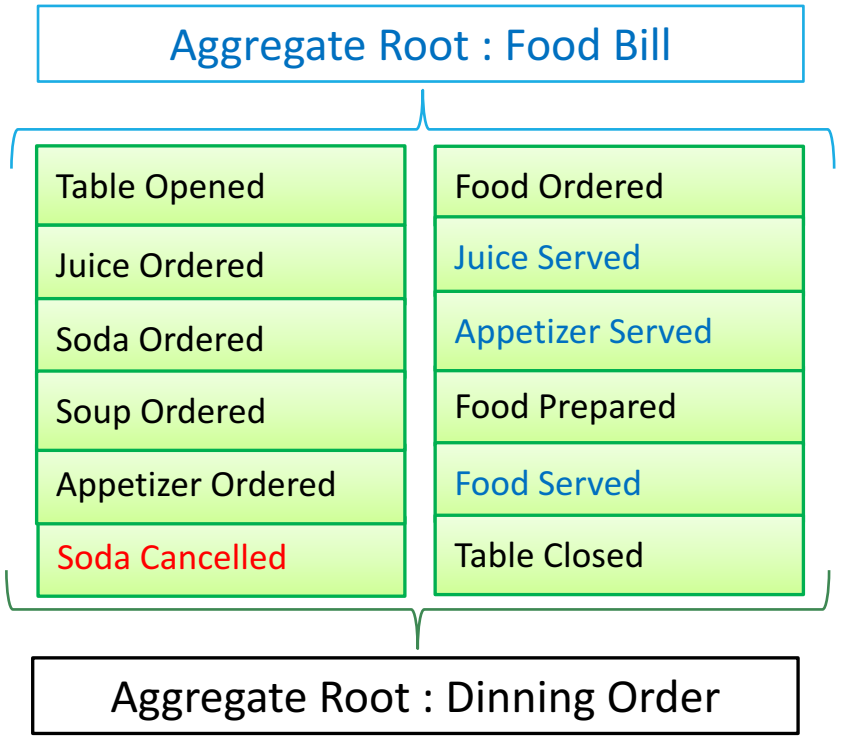
Use Case : Restaurant – Forward Recovery

Domain

The example focus on a concept of a Restaurant which tracks the visit of an individual or group to the Restaurant. When people arrive at the Restaurant and take a table, a table is opened.

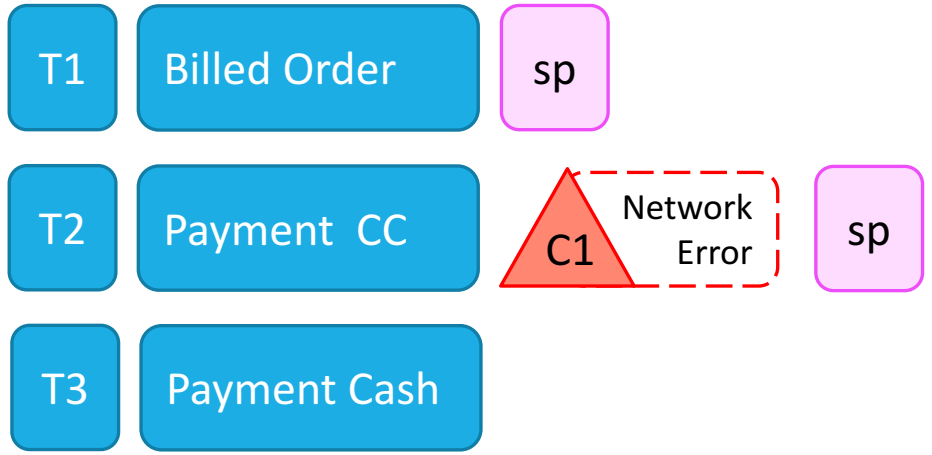
They may then order drinks and food. Drinks are served immediately by the table staff, however food must be cooked by a chef. Once the chef prepared the food it can then be served.

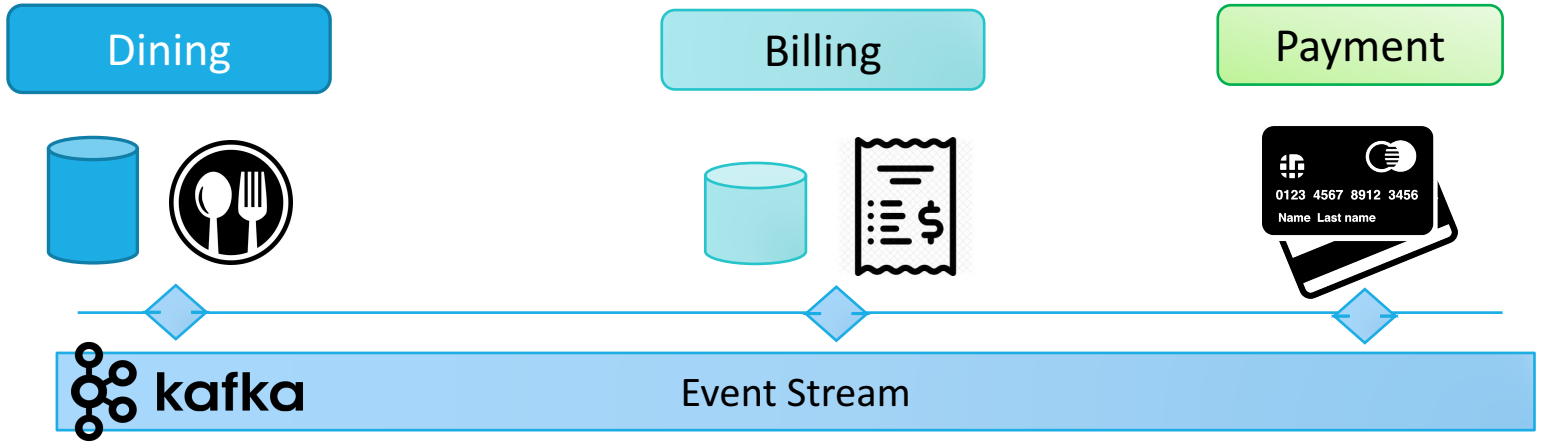
Source: http://cqrs.nu/tutorial/cs/01-design



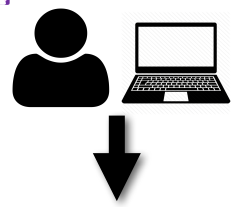
Transaction doesn't rollback if one payment method is failed. It moves forward to the NEXT one.

T_1 (sp) T_2 (sp) T_3 (sp)



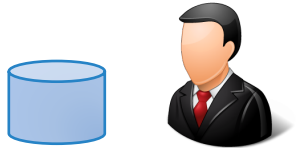
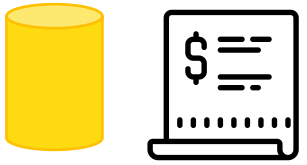


Use Case : Shopping Site – Event Sourcing / CQRS



Commands are End-User interaction with the App and based on the commands (Actions) Events are created. These Events includes both Domain Events and Integration Events. Event Sourced Aggregates will be derived using Domain Events. Each Micro Service will have its own separate Database. Depends on the scalability requirement each of the Micro Service can be scaled separately. For Example. Catalogue can be on a 50 node cluster compared to Customer Micro Service.

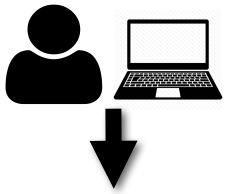
Customer Journey thru Shopping Process

Commands 2				ES Aggregate 4
• Search Products • Add Products • Update Products	• Add to Cart • Remove Item • Update Quantity	• Process Order • Select Address • Select Delivery Mode	• Proceed for Payment • Confirm Order for Payment • Cancel Order	• Customer • Shop.. Cart • Order
Catalogue	Shopping Cart	Customer	Order	Payment
				
Events 3				
• Product Added • Product Updated • Product Discontinued	• Item Added • Item Removed / Discontinued • Item Updated • Order Initiated	• Address Selected • Delivery Mode Selected • Order Created	• Payment Initiated • Order Cancelled • Order Confirmed	• OTP Send • Payment Approved • Payment Declined

Microservices

The purpose of this example is to demonstrate the concept of ES / CQRS thru Event Storming principles.

Use Case : Movie Booking – Event Sourcing / CQRS



Commands are End-User interaction with the App and based on the commands (Actions) Events are created. These Events includes both Domain Events and Integration Events. Event Sourced Aggregates will be derived using Domain Events. Each Micro Service will have its own separate Database. Depends on the scalability requirement each of the Micro Service can be scaled separately. For Example. Theatre can be on a 50 node cluster compared to Food Micro Service.

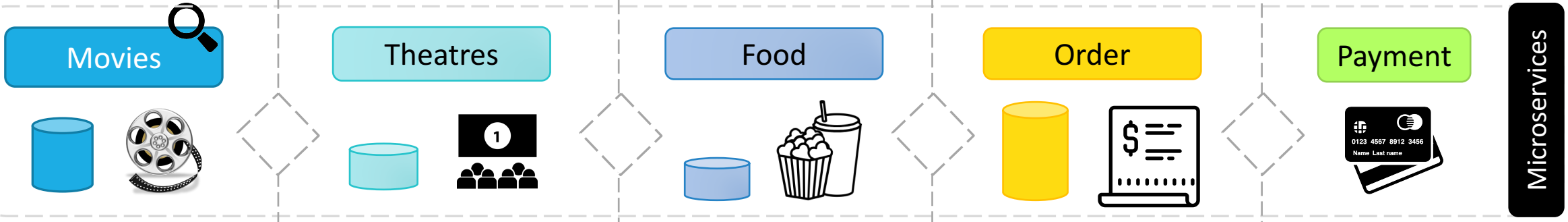
Customer Journey thru booking Movie Ticket

Commands 2

• Search Movies • Add Movies • Update Movies	• Select Movie • Select Theatre / Show • Select Tickets • Process Order	• Select Food • Food Removed • Skip Food • Process Order	• Proceed for Payment • Confirm Order for Payment • Cancel Order

ES Aggregate 4

- Theatre
- Show
- Order

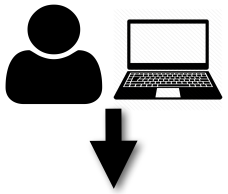


Events 3

• Movie Added • Movie Updated • Movie Discontinued	• Movie Added • Theatre / Show Added • Tickets Added • Order Initiated	• Popcorn Added • Drinks Added • Popcorn Removed • Order Finalized	• Payment Initiated • Order Cancelled • Order Confirmed	• OTP Send • Payment Approved • Payment Declined

The purpose of this example is to demonstrate the concept of ES / CQRS thru Event Storming principles.

Use Case : Movie Streaming – Event Sourcing / CQRS



Commands are End-User interaction with the App and based on the commands (Actions) Events are created. These Events includes both Domain Events and Integration Events. Event Sourced Aggregates will be derived using Domain Events. Each Micro Service will have its own separate Database. Depends on the scalability requirement each of the Micro Service can be scaled separately. For Example. Theatre can be on a 50 node cluster compared to Food Micro Service.

Customer Journey thru Streaming Movie / TV Show

Commands 2

- Search Movies
- Add Movies
- Update Movies

- Request Streaming
- Start Movie Streaming
- Pause Movie Streaming

- Validate Streaming License
- Validate Download License

- Subscribe Monthly
- Subscribe Annually

ES Aggregate 4

- Stream List
- Favorite List

Discovery | **Streaming** | **License** | **Subscription** | **Payment**





Microservices

Events 3

- Movie Added
- Movie Updated
- Movie Discontinued

- Streaming Requested
- Streaming Started
- Streaming Paused
- Streaming Done

- Streaming Request Accepted
- Streaming Request Denied

- Monthly Subscription Added
- Yearly Subscription Added

- Payment Approved
- Payment Declined

The purpose of this example is to demonstrate the concept of ES / CQRS thru Event Storming principles.

User Case : Restaurant Dining Example – Customer Journey

1 Processes

When people arrive at the Restaurant and take a table, a Table is opened. They may then order drinks and food. Drinks are served immediately by the table staff, however food must be cooked by a chef. Once the chef prepared the food it can then be served. Table is closed when the bill is prepared.

Customer Journey thru Dinning Processes

2 Commands

- Add Drinks
- Add Food
- Update Food

- **Open Table**
- Add Juice
- Add Soda
- Add Appetizer 1
- Add Appetizer 2
- Remove Soda
- Add Food 1
- Add Food 2
- Place Order
- **Close Table**

- Serve Drinks
- Prepare Food
- Serve Food

- **Prepare Bill**
- Process Payment

4 ES Aggregate

- Dinning Order
- Billable Order

Food Menu | Dining | Kitchen | Order | Payment

Microservices

3 Events

- Drinks Added
- Food Added
- Food Updated
- Food Discontinued

- Table Opened
- Juice Added
- Soda Added
- Appetizer 1 Added
- Appetizer 2 Added
- Remove Soda
- Food 1 Added
- Food 2 Added
- Order Placed
- Table Closed

- Juice Served
- Soda Served
- Appetizer Served
- Food Prepared
- Food Served

- Bill Prepared
- Payment Processed

- Payment Approved
- Payment Declined
- Cash Paid

The purpose of this example is to demonstrate the concept of ES / CQRS thru Event Storming principles.

Summary – Saga and Event Storming

- Scalable Distributed Transactions
- Local SAGA & Distributed SAGA
- Saga Execution Container
- Domain and Integration Events
- Let-it-Crash Design Pattern and Erlang Philosophy

ESP Diagrams

User Journey : _____

	Actions 2	Micro Service	Events 3	ES Aggregate 4
Process 1 Name				
Process 2 Name				
Process 3 Name				

1. Process
2. Commands
3. Events
4. ES Aggregate
5. Projections

FACTS

References

1. Lewis, James, and Martin Fowler. “Microservices: A Definition of This New Architectural Term”, March 25, 2014.
2. Miller, Matt. “Innovate or Die: The Rise of Microservices”. *e Wall Street Journal*, October 5, 2015.
3. Newman, Sam. *Building Microservices*. O’Reilly Media, 2015.
4. Alagarasan, Vijay. “Seven Microservices Anti-patterns”, August 24, 2015.
5. Cockcroft, Adrian. “State of the Art in Microservices”, December 4, 2014.
6. Fowler, Martin. “Microservice Prerequisites”, August 28, 2014.
7. Fowler, Martin. “Microservice Tradeoffs”, July 1, 2015.
8. Humble, Jez. “Four Principles of Low-Risk Software Release”, February 16, 2012.
9. Ketan Gote, May 22, 2017 – Zuul Edge Server
10. Ketan Gote, May 22, 2017 – Ribbon, Hysterix using Spring Feign
11. Ketan Gote, May 22, 2017 – Eureka Server with Spring Cloud
12. Ketan Gote, May 20, 2017 – Apache Kafka, A Distributed Streaming Platform
13. Araf Karsh Hamid, August 7, 2016 – Functional Reactive Programming
14. Araf Karsh Hamid, July 30, 2016 – Enterprise Software Architectures
15. Araf Karsh Hamid, April 28, 2015 – Docker and Linux Containers
16. Araf Karsh Hamid, Oct 6, 2017 – Micro Services Architecture

References

Domain Driven Design

17. Oct 27, 2012 What I have learned about DDD Since the book. By Eric Evans
18. Mar 19, 2013 Domain Driven Design By Eric Evans
19. May 16, 2015 Microsoft Ignite: Domain Driven Design for the Database Driven Mind
20. Jun 02, 2015 Applied DDD in Java EE 7 and Open Source World
21. Aug 23, 2016 Domain Driven Design the Good Parts By Jimmy Bogard
22. Sep 22, 2016 GOTO 2015 – DDD & REST Domain Driven API's for the Web. By Oliver Gierke
23. Jan 24, 2017 Spring Developer – Developing Micro Services with Aggregates. By Chris Richardson
24. May 17. 2017 DEVOXX – The Art of Discovering Bounded Contexts. By Nick Tune

Event Sourcing and CQRS

25. Nov 13, 2014 GOTO 2014 – Event Sourcing. By Greg Young
26. Mar 22, 2016 Spring Developer – Building Micro Services with Event Sourcing and CQRS
27. Apr 15, 2016 YOW! Nights – Event Sourcing. By Martin Fowler
28. May 08, 2017 When Micro Services Meet Event Sourcing. By Vinicius Gomes

References

29. MSDN – Microsoft https://msdn.microsoft.com/en-us/library/dn568103.aspx
30. Martin Fowler : CQRS – http://martinfowler.com/bliki/CQRS.html
31. Udi Dahan : CQRS – http://www.udidahan.com/2009/12/09/clarified-cqrs/
32. Greg Young : CQRS - https://www.youtube.com/watch?v=JHGkaShoyNs
33. Bertrand Meyer – CQS - http://en.wikipedia.org/wiki/Bertrand_Meyer
34. CQS : http://en.wikipedia.org/wiki/Command–query_separation
35. CAP Theorem : http://en.wikipedia.org/wiki/CAP_theorem
36. CAP Theorem : http://www.julianbrowne.com/article/viewer/brewers-cap-theorem
37. CAP 12 years how the rules have changed
38. EBay Scalability Best Practices : http://www.infoq.com/articles/ebay-scalability-best-practices
39. Pat Helland (Amazon) : Life beyond distributed transactions
40. Stanford University: Rx https://www.youtube.com/watch?v=y9xudo3C1Cw
41. Princeton University: SAGAS (1987) Hector Garcia Molina / Kenneth Salem
42. Rx Observable : https://dzone.com/articles/using-rx-java-observable

References

43. Greg Young. What is a Domain Event? http://codebetter.com/gregyoung/2010/04/11/what-is-a-domain-event/
44. Jan Stenberg. Domain Events and Eventual Consistency https://www.infoq.com/news/2015/09/domain-events-consistency
45. Jimmy Bogard. A better domain events pattern https://lostechies.com/jimmybogard/2014/05/13/a-better-domain-events-pattern/
46. Vaughn Vernon. Effective Aggregate Design Part II: Making Aggregates Work Together http://dddcommunity.org/wp-content/uploads/files/pdf_articles/Vernon_2011_2.pdf
47. Jimmy Bogard. Strengthening your domain: Domain Events*https://lostechies.com/jimmybogard/2010/04/08/strengthening-your-domain-domain-events/ *
48. Tony Truong. Domain Events Pattern Example http://www.tonytruong.net/domain-events-pattern-example/
49. Udi Dahan. How to create fully encapsulated Domain Models http://udidahan.com/2008/02/29/how-to-create-fully-encapsulated-domain-models/
50. Udi Dahan. Domain Events – Take 2 http://udidahan.com/2008/08/25/domain-events-take-2/
51. Udi Dahan. Domain Events – Salvation http://udidahan.com/2009/06/14/domain-events-salvation/
52. Jan Kronquist. Don't publish Domain Events, return them! https://blog.jayway.com/2013/06/20/dont-publish-domain-events-return-them/
53. Cesar de la Torre. Domain Events vs. Integration Events in DDD and microservices architectures https://blogs.msdn.microsoft.com/cesardelatorre/2017/02/07/domain-events-vs-integration-events-in-domain-driven-design-and-microservices-architectures/

References – Micro Services – Videos

54. Martin Fowler – Micro Services : https://www.youtube.com/watch?v=2yko4TbC8cI&feature=youtu.be&t=15m53s
55. GOTO 2016 – Microservices at NetFlix Scale: Principles, Tradeoffs & Lessons Learned. By R Meshenberg
56. Mastering Chaos – A NetFlix Guide to Microservices. By Josh Evans
57. GOTO 2015 – Challenges Implementing Micro Services By Fred George
58. GOTO 2016 – From Monolith to Microservices at Zalando. By Rodrigue Scaefer
59. GOTO 2015 – Microservices @ Spotify. By Kevin Goldsmith
60. Modelling Microservices @ Spotify : https://www.youtube.com/watch?v=7XDA044tl8k
61. GOTO 2015 – DDD & Microservices: At last, Some Boundaries By Eric Evans
62. GOTO 2016 – What I wish I had known before Scaling Uber to 1000 Services. By Matt Ranney
63. DDD Europe – Tackling Complexity in the Heart of Software By Eric Evans, April 11, 2016
64. AWS re:Invent 2016 – From Monolithic to Microservices: Evolving Architecture Patterns. By Emerson L, Gilt D. Chiles
65. AWS 2017 – An overview of designing Microservices based Applications on AWS. By Peter Dalbhanjan
66. GOTO Jun, 2017 – Effective Microservices in a Data Centric World. By Randy Shoup.
67. GOTO July, 2017 – The Seven (more) Deadly Sins of Microservices. By Daniel Bryant
68. Sept, 2017 – Airbnb, From Monolith to Microservices: How to scale your Architecture. By Melanie Cubula
69. GOTO Sept, 2017 – Rethinking Microservices with Stateful Streams. By Ben Stopford.
70. GOTO 2017 – Microservices without Servers. By Glynn Bird.

5.10

User Journey : _____

	Actions 2	Micro Service	Events 3	ES Aggregate 4
Process 1 Name				
Process 2 Name				
Process 3 Name				

Thank you

Araf Karsh Hamid :
Co-Founder / CTO MetaMagic Global Inc. NJ, USA
araf.karsh@metamagic.in | araf.karsh@gmail.com
USA: +1 (973) 969-2921
India: +91.999.545.8627
Skype / LinkedIn / Twitter / Slideshare : arafkarsh
http://www.slideshare.net/arafkarsh
https://www.linkedin.com/in/arafkarsh/

*A **Micro Service** will have its own Code Pipeline for build and deployment functionalities and it's scope will be defined by the Bounded Context focusing on the Business Capabilities and the interoperability between Micro Services will be achieved using (asynchronous) message based communication.*